Colored-Object Programming : Inheritance by Dimensions

Henry J. Borron

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Colored-Object Programming : Inheritance by Dimensions

Henry J. Borron

N° 2878
Avril 1996

THÈME 2

Colored-Object Programming : inheritance by dimensions.

Henry J. Borron *

Programme 2 — Génie Logiciel et Calcul Symbolique
Action LeTool

Rapport de Recherche N° 2878 — Avril 1996 — 122 pages

Abstract. This paper is about colored object programming. It focuses on a rigorous approach of class inheritance using the color graph formalism. This one abstracts the whole behaviour of a class instances in a N-dimensions space.

The paper thus relates a color graph of one class with the color graphs of its superclasses. This is done first at the abstract level of transitions, and then at the implementation level (memory representations are attached to states, pre- and post-methods to transitions), each time by devising and extending the inheritance rules in one given color graph.

This work does not depend on the visual formalim itself : it may well be incorporated in state-transition formalisms based on insideness instead of connectedness (i.e. formalisms derived from statecharts).

Keywords. Colored object, state, transition, mixin, inheritance, combination, language design, visual formalism, object oriented programming, abstraction, modularity, monotonicity, linearity, cleanness.

(Résumé : tsvp)

* borron@chris.inria.fr

INRIA, Unité de Recherche de Sophia-Antipolis
2004, route des Lucioles. BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 — Télécopie : (33) 93 65 77 65

Programmation par Objets Colorés : héritage par dimensions.

Résumé. Ce papier a trait à la programmation par objets colorés. Il est centré sur une approche rigoureuse de l'héritage des classes via l'utilisation du formalisme des graphes colorés. Celui-ci abstrait le comportement global des instances d'une classe dans un espace à N-dimensions.

Le papier relie donc le graphe coloré d'une classe aux graphes colorés de ses superclasses. Ceci est fait d'abord au niveau abstrait des transitions, et ensuite au niveau de l'implémentation (des représentations en mémoire sont attachées aux états ; des pré- et post-méthodes, aux transitions), à chaque fois par élaboration et extension des règles d'héritage dans un graphe coloré donné.

Ce travail ne dépend pas du formalisme visuel lui-même. Il peut être incorporé aux formalismes visuels par états et transitions fondés sur l'inclusion au lieu de la connexion.

Mots-clés. Objet coloré, état, transition, mixin, héritage, combinaison, conception de langage, formalisme visuel, programmation par objets, abstraction, modularité, monotonicité, linéarité, propreté.

Colored-Object Programming : Inheritance by Dimensions.

1. INTRODUCTION

1.1 GOAL OF THE PAPER

This paper is the third of a sery of three on Colored Object Programming (COP, for short)[1]. It focuses on a rigorous approach of **class inheritance** using the formalism presented in the first companion paper [Borron, 1996d] and results obtained about mixin behaviours (i.e. independent supplementary behaviours) in the second companion paper [Borron, 1996e]. It also strongly relates to a forthcoming paper describing the design and properties of our linearization algorithm [Borron, 1996x].

Let's substantiate this. The goal of COP is to push the idea of object to its ultimate. Consequently, the objects considered in COP have **states** ; and transitions exist between these states. The **transitions** in question (termed "regular transitions") are triggered by external events (messages or generic function calls). At this external level, no consideration at all is given to methods nor memory representations[2] : these are really second citizens in our approach, yet they implicitely exist for supporting the states and transitions. Given a class and its instances, the proposed formalism abstracts the **WHOLE** behaviour of the considered instances (the whole behaviour, and not an increment) using these two concepts of states and transitions.

We can now rephrase our goal into more precise questions :

(1) because a class and what is usually termed its superclasses are all given an abstract description of the WHOLE behaviour of their instances, a first question is : *«How does the abstract global behaviour defined for one class of objects relates to the abstract global behaviour defined for "its superclasses" ?»* ;

(2) considering implementations, another question is : *«What is the relationship between an abstract global behaviour and its implementation, i.e. its methods and memory representations ?»* ;

(3) the last question is : *«How does a class inherits (parts of) its implementation from its superclasses ?».*

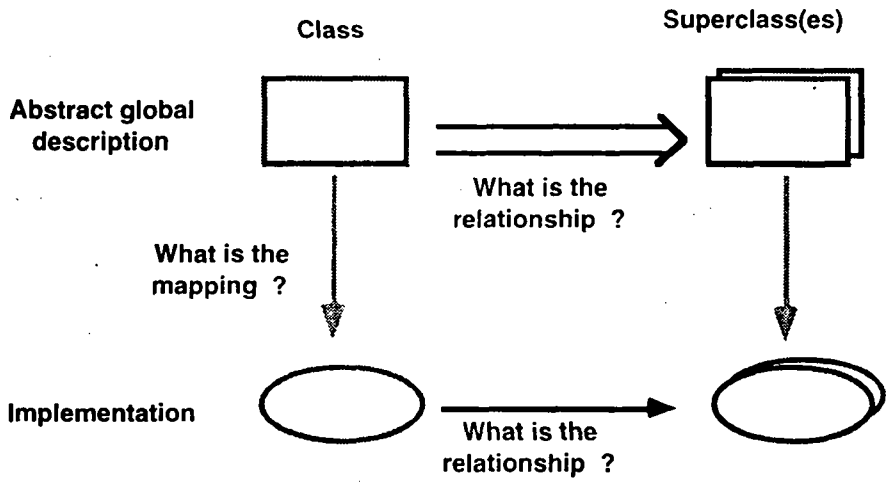


Figure 1.

The problem of inheritance is thus twofold : one is at the level of states and transitions (abstract or specification level) ; one at the level of methods and memory representations (concrete or implementation level). In addition to that, we have

[1] The original version of this paper was written in October 1995. Since then, the description has been made much more detailed.

[2] a memory representation is made of a number of cells, i.e. instance variables [Smalltalk wording] or slots [CLOS wording]

to define how we attach an implementation to an abstract description of states and transitions (such a description is termed a "color graph").

Basically, for each inheritance question, our approach consists in :
(a) devising inheritance rules in ONE given color graph (these rules are termed the LOCAL inheritance rules) ;
(b) extending them to a HIERARCHY of color graphs (hence, the CLASS inheritance rules).

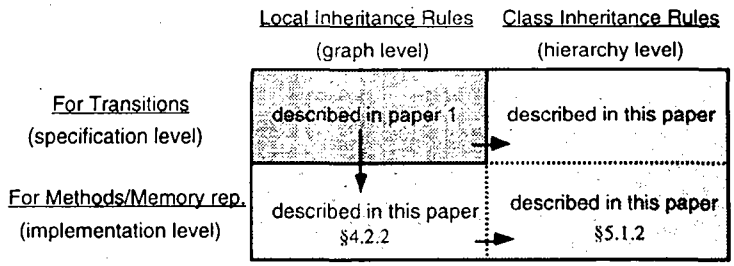


Figure 2.

Two points are worth to be mentioned here for guiding the comprehension :

(1) the LOCAL inheritance rules at the ABSTRACT level are in fact supposed to be known in this paper : these are the rules for inheriting TRANSITIONS in ONE given color graph (they have been described in companion paper n°1) ;

(2) the key point for inheritance is going to be the decomposition into **dimensions** (a color graph describes the behaviour of a class of objects in a N-dimension space).

1.2 PLAN OF THE PAPER

This plan is mostly driven by our goals and approach as defined above, the other factor being the necessary information to give about our formalism. Besides an introductory part, it can be separated into three main parts :
— Part A is devoted to the inheritance of transitions (sections 3, 4 and 5) ;
— Part B focuses on the inheritance of implementations, i.e. memory representations and methods (sections 6 and 7) ;
— Part C generalizes the work done previously with, notably, systematic class level combination and multiple dispatch (sections 8, 9 and 10).

Here is the detailed plan :

• Section 2 is devoted to a quick presentation of our formalism to enable the reading of this paper to those that haven't gone through companion papers n°1 and 2. LOCAL inheritance rules for TRANSITIONS are stated briefly in this section. An example of color graph is given to ease the comprehension.

-------- PART A : INHERITANCE OF TRANSITIONS --------

• Section 3 first presents the basic idea of our approach using the previous example for informal explanations, then it focuses on the key concepts for inheritance.

• Section 4 is about the LOCAL inheritance of TRANSITIONS. Although supposed to be known from companion paper n° 1, this form of inheritance is rephrased here to provide a solid basis for the rest.

• Section 5 shows how the local inheritance rules for transitions can be generalized for handling CLASS inheritance for transitions (in a HIERARCHY of classes, i.e. of color graphs). This section proposes operators for composing dimensions or for adding dimensions in one point. This last aspect leads to mixins.

-------- PART B : INHERITANCE OF IMPLEMENTATIONS --------

• Section 6 describes LOCAL implementations (in a given color graph). This has two aspects :
- the mapping of a color graph to an implementation (memory representations are attached to states, pre- and post-methods to transitions) ;
- LOCAL inheritance for IMPLEMENTATIONS : how a memory representation or a pre/post-method attached in a given node of a color graph may be inherited in one another node of the same color graph ? Obviously enough, a connection is made here with the local inheritance of transitions.

• Section 7 describes CLASS implementations (in a hierarchy of classes, i.e. of color graphs). This is done very progressively first considering superclasses only, then adding a local description. When finally considering all ancestors, the case of a DAG structure is analysed in detail. A dimension inherited along several paths is -by default- considered as unique. At this point, the spatial interpretation of inheritance appears crucial. This section proposes, for each dimension, a linearization of classes impacting it. Hence, a discussion on properties of linearization algorithm (a new property, termed congruency, is defined and analysed). The section ends with a new monotonic linearization algorithm.

-------- PART C : GENERALIZATION --------

• Section 8 specifies how a systematic class-level combination style can be used. This contrasts with the previous sections where a systematic masking style was supposed. An important property (termed regularity) is defined in this section. This property enables the organization of the methods found along the dimensions of a hierarchy to be a list of trees (provided that some natural rule is applied).

• Section 9 shows how multiple dispatch can be obtained assuming an extension of the regularity property.

• Section 10 shows that sophisticated combination methods can be obtained without the **send-super** anti-modular construct (**super** in Smalltalk, **call-next-method** in CLOS). It shows that the CLOS-style of combination with qualifiers (ex. : the standard combination mechanism) can be mimicked and generalized, iincluding in case of multiple dispatch. It also shows there is no need in COP for individual methods.

• Section 11 proposes a short summary of the paper.

• Section 12 relates our proposal to other works.

• In the conclusion, we stress the fact that our proposal may well be incorporated into statecharts and derived formalisms, since our own formalism is based on connectedness whereas statecharts are based on insideness. (This was shown in companion paper n° 1).

Note.

(a) This paper (as well as its two companion papers) presents **concepts** and NOT a programming environment. A forthcoming paper will show how a set of friendly tools may support the proposed notation in an interactive way.

(b) Footnotes are meant to provide details to the interested reader : they may be skipped in first reading.

INTRODUCTORY PART

In this part, the color graph formalism is first presented, then it is illustrated with a simple example.

2. COLOR GRAPHS : OVERVIEW

2.1 INTRODUCTION

To abstractly model classes, we use a special form of graphs. Each class is attached one. The graph describes the class considered as a whole : in other words, it exhibits all the external properties of the class, including those inherited from the class ancestors. [This wholistic view contrasts in some way with the traditional OOP view of classes where a class is described as an increment of behaviour. Consider, for example, a *Stack* inheriting –in the OOP sense– from a *Bag*, a *Collection* and an *Object* classes : its color graph will not only express the external properties coded in *Stack*, but also those that are inherited from *Bag*, *Collection* and *Object* (for example, *print*).]

The essence of OOP being to externally consider objects as black boxes and to be interested only in messages flowing between them, the external properties we want to model are messages and their effects on objects. The expression "message sending" is borrowed from Smalltalk, but we use it in a more general sense : for "generic function calls" (CLOS wording), i.e. even when several classes are involved in method selection.

The external effect of a message is to possibly modify the set of messages of its receiver (more generally, of one or several specializers of the corresponding generic function call)[3]. Hence, a color graph is made of nodes and transitions between them : the source node of a transition T models the possibility to receive a message T in a certain state whereas the destination node abstracts the external effect of this message T. We term such a graph a "color graph" [Borron, 1996d], "color" being metaphorically used for expressing that, in our model, objects have states[4].

From the point of view of our model, traditional OOP corresponds to considering but a single state for instances, with all state transitions flowing in and out of it (i.e. being circular and attached to it). Expressed differently, our approach considers several message dictionaries, one per state, whereas traditional OOP considers but one.

2.2 MAIN CONCEPTS

In COP, a class is abstracted as a whole in the form of a **color graph**. Possible (reachable) instance states (or contributions to them) are depicted by nodes (in finite number) ; the existence and effect of external events, by **regular transitions**. Nodes are organized into **essential constructs**. The **skeleton** of a color graph is the structure made by all its constructs (nodes included), i.e. without the regular transitions.

2.2.1) Skeleton

• Possible instance states are described according to one or several **dimensions**[5]. In the first case, the simplest one, each state is represented by a single node (a **color**) ; the corresponding color graph is termed a **c-graph**. In the second case, each state is basically represented by N nodes, one per dimension (each such node is termed a **pigment**) ; the corresponding color graph is termed a **p-graph**. The set of pigments along one same dimension is termed a **scale** (N scales exist). A p-graph may also exhibit **blends** : each one is a node that recursively groups two or more pigments of different scales. As clearly expressed in companion paper n° 1, blends are strictly optional. We term **p-chroma** a pigment or a blend ; we term **chroma** (or, more loosely, node) a color or a p-chroma.

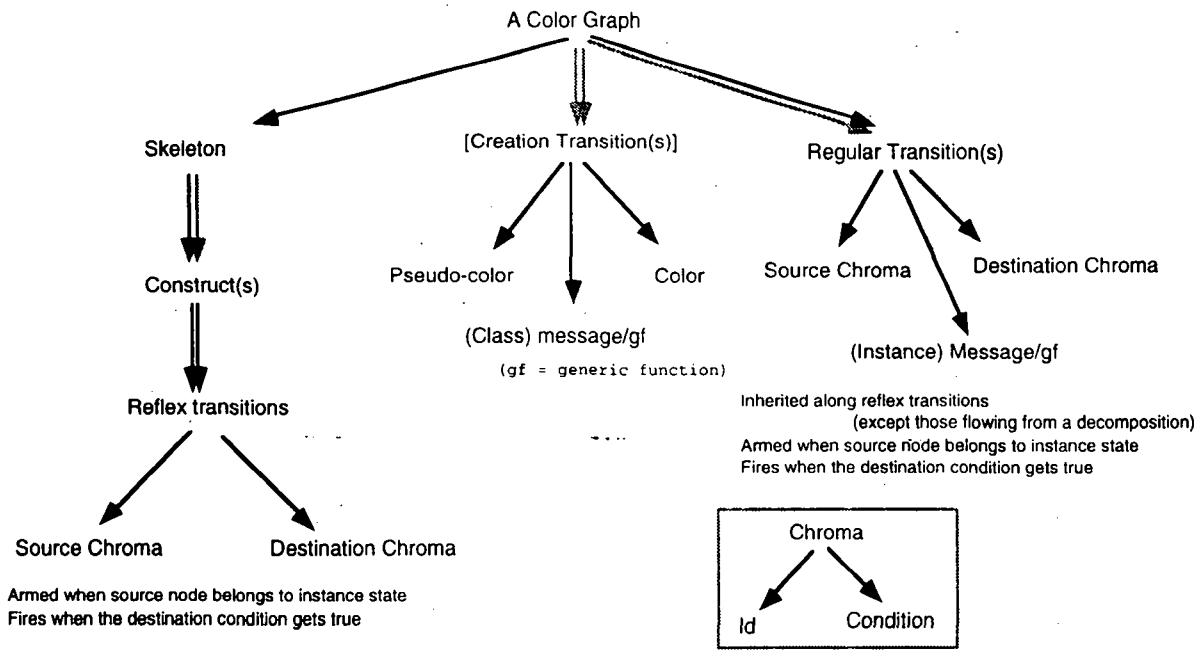
• Each node in a color graph is given a **condition** (evaluated by sending a side-effect free message to the instance) and/or results from an essential construct. Each node has also an **id**.

---

[3] Warning : in the rest of the paper, we usually consider messages (in the narrow sense) and omit to explicitely specify that the shown properties are also valid in case of generic function calls : we do this for conciseness. The user should not forget about that.

[4] Idioms often translate a state as a color. For example, someone may be green with envy, red with shame, etc.

[5] These dimensions form a system of coordinates ("a **palette**") in a N-dimension space.



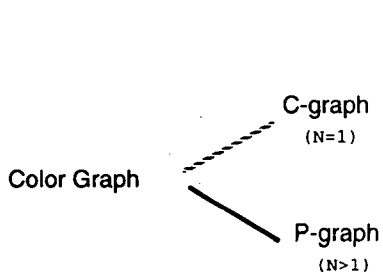
Elements of a Color Graph

Figure 3.

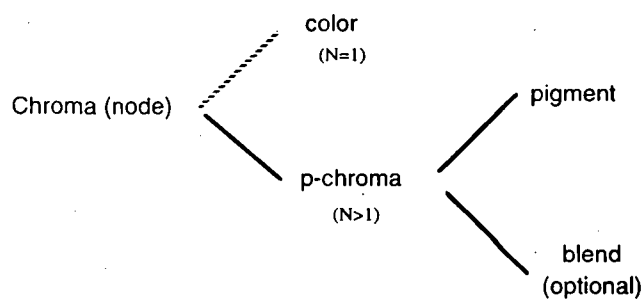
• Essential constructs (**selection, decomposition, conjunction**) are one-level trees of nodes, either diverging (the first two constructs) or converging (the last one), their root node being either of type OR (first construct) or AND (last two). The source node of a selection is qualified as being **ephemere**. One or several destination nodes of a selection may be chosen as **INIT** nodes. The destination node of a conjuction is a blend. In any essential construct, a source node and a destination node are linked by a **reflex transition**.

• A reflex transition is **armed** when its source node participates to the current instance state. An armed reflex transition **fires** when the condition attached to its destination node gets true.

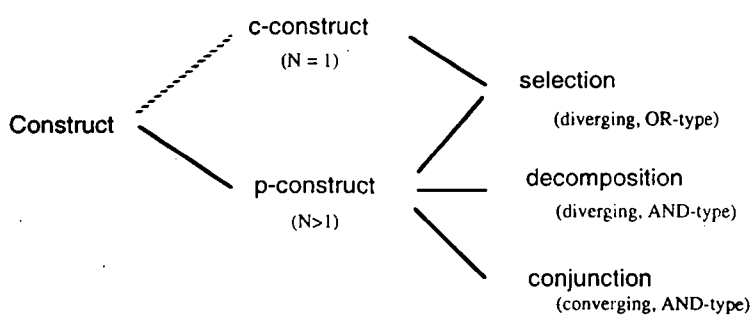
• Given a chroma C, its **ancestor tree** is obtained by recursively gathering the ancestors of this node along the reflex transitions converging to it (except when they originate from the source node of a decomposition).



Typology of Color Graphs



Typology of Chromas



Typology of constructs

Figures 4, 5 & 6.

2.2.2) Regular transitions

• **Regular transitions** fire on an external event (a message or a generic function call : in the first case, a single color graph instance is considered ; in the second case, several ones may be involved).

• Regular transitions are **inherited** along reflex transitions (except along those flowing from a decomposition)[6] : this is what we call the main LOCAL INHERITANCE RULE FOR TRANSITIONS.

• A regular transition is termed **circular** when its destination node is nothing but its source node. A transition is **outcoming** vs. its source node and **incoming** vs. its destination node. A **pure** outcoming (incoming) transition is not circular.

• A usual convention applies : the term "transition" means "regular transition" if not preceded by "reflex".

2.2.3) Mark

The current instance state is marked by a **token** in a 1-dimension color graph, or by a set of N **mini-tokens** in a N-dimension color graph (one per dimension). **Mark** is a generic term for token or mini-token(s). When it fires, a reflex transition moves the mark of its source (selection, conjunction) or part of it (decomposition) to its destination. (A decomposition explodes a token into N mini-tokens.) When it fires, a regular transition moves the mark from its source to its destination.

2.2.4) P otential

A further (non mandatory) refinement of color graphs consists in attaching a value to each (mini-)token. This value, termed **potential**, can be : (1) initialized at (mini-)token creation ; (2) updated when a transition gets fired (using or discarding the arguments of the message) ; (2) tested for conditions evaluation. This gives an abstract body to regular and testing transitions, and substantiates the color graph functioning (ex. : for animation).

2.3 VISUAL ASPECT

A textual syntax (derived from CLOS) and a visual syntax have been elaborated. Let's give some indications about the visual syntax since it will be used extensively in our examples.

A node is pictured as a small bubble : inside the bubble, a name or an integer identifies the node ; close to the bubble, the condition attached to this node is named (inferred conditions are usually not displayed).

Regular graph transitions are represented with named solid thick arrows. **Reflex transitions** are depicted by unnamed thin arrows : using a dotted line for **selections**, a broken line for **conjunctions**. A **decomposition** is not represented using reflex transition arrows, but by a small bar stuck between the root node and the leaf nodes (usually, the root node is only expressed by its name).
These distinctions may be intensified when the medium enables colored drawings : it appears interesting to attribute a same coloration to pigments of a same scale and to reflex transitions outcoming of them ; and to draw all their contours the same way.

Usually, one unnamed arrow ondulates from a special node to an initial color (numbered 1 by default) : it corresponds to the default **creation transition** (*make-instance*)[7]. This special node, a small segment with the class name above it, corresponds to the case an instance is not created or has been destroyed.

Circular transitions may be shown without arrows. They may be further qualified (and shown) as consulting or modifying (default) for coherency checks.

Testing transitions (i.e. regular transitions associated to side-effect free messages used for evaluating conditions), because automatically derived by the system, are usually not represented.

An **INIT destination** node of a selection is shown with a double contour.

---

6 A reflex transition may thus have two possible roles : one dynamic (firing, at run time) ; one static (factorization, before execution).

7 Several creation transitions may exist : in such a case, they should be named (the default one is cancelled).

2.4 A COLOR GRAPH EXAMPLE

The example given here has already been presented in companion paper n°1. Here, it is discussed much less extensively. However, it is shown in both visual and textual form, the latter one being of interest for the rest of the paper.

2.4 .1 Description

Next figure is derived from the source code of the *Person* example given in section 4.2 of [Chambers, 1993][8]. *Person* instances are differentiated into *male* and *female* categories according to the *sex* ; *child*, *teenager* and *adult* categories, according to the *age*. In addition, both criteria are used to distinguish *boy* and *girl* categories.

In terms of COP, two **scales** are used : first one groups **pigments** 2 to 5 ; second one, pigments 6 to 12. **Color** 1 is the **initial state** : it is **decomposed** according to the two scales. **Blends** 13 and 14 result from two **conjunctions**. They may be termed colors since their **degree** is the degree of the color graph, i.e. two. Pigments 2, 4 and 5 (resp. 6, 10, 11, 12) are **basic** pigments in the first scale (resp. in the second one) ; pigments 3, 7, 8, 9 are **ephemere** pigments. Ephemere pigment 7 is the source of a **selection** on three pigments : this is somewhat unusual since selections are quite often based on a boolean value.

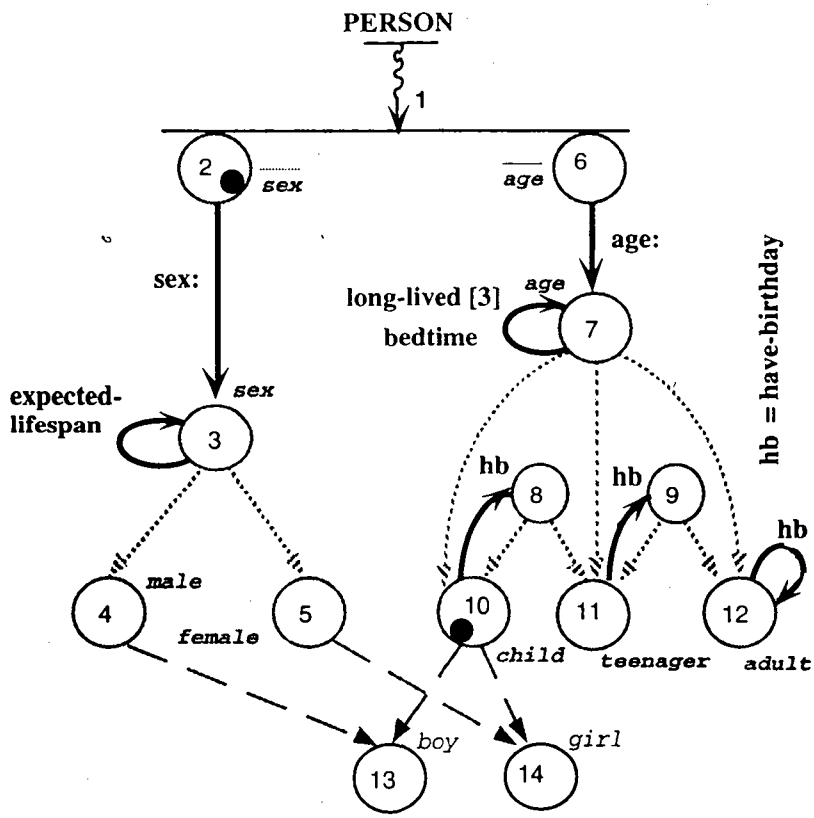


Figure 7.

Next figure shows the textual syntax for declaring the whole set of chromas (also termed the **skeleton**). (In the following, we may well use condition names to refer to a chroma instead of its number : for example, pigment 7 will also be termed *age*.)

[8] For the purpose of the demonstration, a slight difference exists to create an instance of *Person* : the cited paper proposes a *make-person* method with values for *sex:* and *age:*. The corresponding creation transition is not represented in figure 7.

```
(defcolorgraph PERSON
     (  (color     1 :decomposition 2 6)
        (pigment   2 :test-not sex)
        (pigment   3 :test sex)
        (pigment   3 :selection 4 5)
        (pigment   4 :test male)
        (pigment   5 :test female)
        (pigment   6 :test-not age)
        (pigment   7 :test age)
        (pigment   7 :selection 10 11 12)
        (pigment   8 :selection 10 11)
        (pigment   9 :selection 11 12)
        (pigment  10 :test child)
        (pigment  11 :test teenager)
        (pigment  12 :test adult)
        (color    13 :conjunction 4 10)
        (color    13 :test boy)
        (color    14 :conjunction 5 10)
        (color    14 :test girl)))
```

Figure 8.

Messages *sex:* and *age:* are used for **initialization**. Such messages modifies the instance state. Sending an *expected-lifespan* to one instance does not change its state : *expected-lifespan* is thus represented as a **circular transition**. *Expected-lifespan* cannot be sent when the *sex* is unknown (the value it returns depends on whether the instance is *male* or *female*) : one possibility is to attach an *expected-lifespan* circular transition to both *male* and *female* pigments. In fact, given the **local inheritance rule for transitions**, it is equivalent and simpler to factorize this circular transition in the ephemere *sex* pigment (the result is what we named an **i-circular** transition, each individual transition it replaces being itself circular).

For similar reasons, *bedtime* is represented by an "i-circular" transition attached to the *age* pigment.

The *have-birthday* message is kind of special since, by incrementing the *age*, it makes an instance stay in its pigment (ex : *child*) or move to the next one (ex: *teenager*) except when in the *adult* pigment. No symmetry exists we can take advantage of. To model this situation, two **selections** are used (for the *have-birthday* transitions outcoming from the *child* and *teenager* pigments) plus one circular transition (for the *have-birthday* transition outcoming from the *adult* pigment). This makes pigments *child*, *teenager* and *adult* to inherit from two or three nodes.
Long-lived — which tells if a given instance has gone farther its *expected-lifespan* — is represented by a **constrained** "i-circular" transition attached to the *age* pigment. This is because both the *sex* and the *age* should be known : one condition is captured in having the transition attached to the *age* pigment ; one, in having a **clause** mentioning the *sex* pigment (number 3). (The opposite choice is perfectly acceptable.) *Long-lived* is valid for any fully initialized state, among which *boy* and *girl* : the notation we used avoids the enumeration of all these states (6 stable ones).[9]

user-defined regular transitions

```
(deftransition sex:                ( (p PERSON (2 3)) s))
(deftransition expected-lifespan   ( (p PERSON (3 3)) ))

(deftransition age:                ( (p PERSON (6 7)) a))
(deftransition bedtime             ( (p PERSON (7 7)) ))
(deftransition have-birthday       ( (p PERSON (10 8) (11 9) (12 12)) ))

(deftransition long-lived          ( (p PERSON 3.(7 7)) ))
```

Figure 9.

Testing transitions are not shown. These are *sex*, *male*, *female* (for the first scale) ; *age*, *child*, *teenager*, *adult* (for the second scale) ; *boy* and *girl* (valid not only for blends *boy* and *girl*, but for all fully initialized instances). Due to the local

[9] In our example, the *long-lived* transition does not affect the instance state : if it were the case, *long-lived* could still be represented by one constrained pure outcoming transition if only one contribution to the instance state was changed ; two constrained pure outcoming transitions would be used otherwise (one from pigment *sex*, one from pigment *age*), thus examplifying what we call a **composite transition** or (pure) **multi-micro-transition**.

inheritance rule, *sex* is attached to pigments 2 and 3 ; *male* and *female*, to pigment 3 ; *age*, to pigments 6 and 7 ; *child*, *teenager* and *adult*, to pigment 7 ; *boy* and *girl*, to pigments 3 and/or 7 (both are required under some form)[10].

As a matter of fact, a number of these testing transitions are in fact **constant transitions** (i.e. transitions that deliver a constant answer in a given chroma) : for example, *sex* can be (externally) asked in pigments 2, 4 and 5, yielding constant answers (resp. false, true, true). All constant answers (method bodies) are automatically generated by the underlying system.

The user will only be required to provide methods for *male* and *female* (automatically sent in pigment 3 for choosing between pigments 4 and 5) and for *child*, *teenager* and *adult* (automatically sent in pigment 7 for choosing between pigments 10, 11 and 12).

testing transitions (automatically defined)

```
(deftransition sex                ( (p PERSON (2 2) (3 3)) ))
(deftransition male               ( (p PERSON (3 3)) ))
(deftransition female             ( (p PERSON (3 3)) ))

(deftransition age                ( (p PERSON (6 6) (7 7)) ))
(deftransition child              ( (p PERSON (7 7)) ))
(deftransition teenager           ( (p PERSON (7 7)) ))
(deftransition adult              ( (p PERSON (7 7)) ))

(deftransition boy                ( (p PERSON (3 3).(7 7)) ))
(deftransition girl               ( (p PERSON (3 3).(7 7)) ))
```

Figure 10.

The **creation** of a *Person* instance is done using the default possibility (name : *make-instance* ; initial state : 1).

default creation transition (automatically defined)

```
(deftransition make-instance      ( (p PERSON (- 1)) ))
```

Figure 11.

Acceptable messages for a fully uninitialized instance are *sex:* and *age:*. Acceptable messages for a fully initialized instance (like *boy* or *girl*) are *sex, male, female, expected-lifespan, age, bedtime, have-birthday, child, teenager, adult, long-lived, boy, girl.* A partially initialized instance accepts *age, bedtime, have-birthday, child, teenager, adult,* and *sex:* (if *age* is initialized) ; *sex, male, female, expected-lifespan* and *age:* (if *sex* is initialized).

A *boy* (or a *girl*) is **dependent** on *sex* and *age* pigments. The *sex* is not supposed to change ; but, the *age* may (cf. the *have-birthday* transitions). Hence *boy* and *girl* —as well as *child, teenager* and *adult*— depend on a modification of *age* by *have-birthday.* So, if *have-birthday* occurs, the state of the instance is checked and —if necessary— computed again. For example, a *boy* may become a (*male teenager*). .This is done by first moving the **mini-token** which is involved in the *have-birthday* inherited transition (*age* mini-token) to the source chroma of this transition (here, *child*) ; this invalidates the blend *boy* : the mini-token which is not involved in the inherited transition (*sex* mini-token) is thus moved back to its original pigment (here, *male*). Then the transition gets **fired** : the *age* mini-token remains in *child* or moves to *teenager.* In the first case, the conjunction (*male child*) fires : *boy* defines again the instance state. In the second case, the instance state is defined by (*male teenager*).

2.4 .2 Variations... and their impacts on complexity and inheritance rules

To introduce additional concepts and remarks, let's discuss a bit more deeply the above *have-birthday* and *long-lived* modellings. Two important issues will appear (cognitive complexity, inheritances rules).

---

[10] The *boy* and *girl* **multi-micro-transitions** are automatically installed by the underlying system at ephemere pigment 3 (*sex*) for the first scale, and –a priori– at ephemere pigment 7 (*age*) and 8 for the second scale ; since the destinations of the selection in 8 (i.e pigments 10 and 11) are all part of the destinations of the selection in 7 (i.e pigments 10, 11 and 12), the ephemere pigment 8 is automatically eliminated. Given that *boy* and *girl* are i-circular, the underlying system may simplify the corresponding multi-micro-transitions (in 3 and 7) into constrained i-circular transitions (either in 3 or 7).

1) Variations

a) Concerning the *have-birthday* transition, two variations deserve some comments.

-> Because of this transition, pigments *child*, *teenager* and *adult* inherit from two or three nodes. A simplification of the **ancestor trees** of these three pigments is possible at the expense of less precision. Instead of one or two possible destinations, each *have-birthday* transition may be considered to have three (*child*, *teenager* and *adult*) : nodes 8 and 9 thus disappear. Next figure shows the simplified color graph.

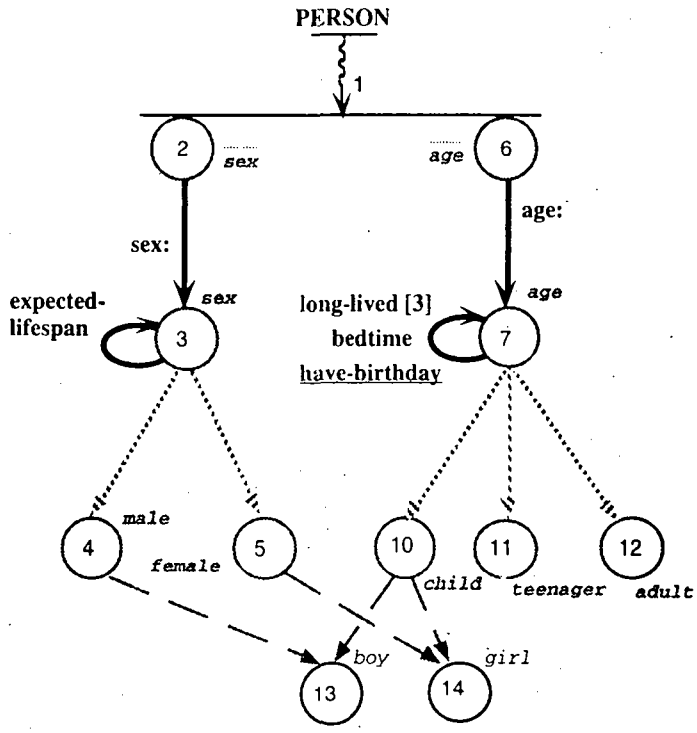


Figure 12.

Less static information is captured into the simplified color graph. This modelling is obviously less efficient (notably in the *adult* case), yet perfectly correct vs. the effective destination. (In each case, after having been modified by incrementation, the *age* contribution to the instance state will be tested against the three possible pigments.) Advantage : this modelling is open to factorization. The three *have-birthday* transitions flowing to the ephemere pigment *age* get naturally replaced by a single circular transition attached to this same pigment. This factorizing transition is not i-circular. To avoid a possible confusion, it is marked as "**g-circular**" : "g" means the actual destination is to be dynamically chosen among the group of the selection destinations[11]. Note this variation is obtained without modifying the cognitive principles underlying the design of color graphs. In the visual representation, a g-circular transition is underlined.

-> Another possibility to model the situation would be to use two outcoming *have-birthday* transitions in pigments *child* and *teenager*, thus violating our **"unique destination" principle**. (See figure 19 for an excerpt of the *Person* graph.) This principle is meant for regularity and simplicity, two important factors for diminishing the cognitive load imposed on a programmer. It states a regular transition outcoming from a given node should have systematically but one destination : in case several ones exist, this principle is obeyed via the use of a selection.
This principle may well in fact be relaxed in situations like the one exposed, this because the ephemere nodes in question (8 and 9) are not attached any transition except the testing ones : being not used for factorization facilities, they somehow clutter the color graph with extra nodes and reflex transitions that can all be generated automatically (for internal consistency) without any loss in power. Not considering the multiplication of regular transitions, this abandon simplifies the visual and textual representations of the color graph. The ancestor trees of the destination chromas of the suppressed selections (here, *child*, *teenager* and *adult*) are accordingly simplified[12].

---

11 By default, a transition factorized in an ephemere node is considered to be g-circular. Explicit declarations are possible by attaching an **:updating** or **:consulting** keyword to the transition declaration. Note the selector syntax may also be used for discrimination.

12 This may impact the placement of representation and methods (cf. the local inheritance rules for these items in a subsequent section) : hence, the application designer may well decide to re-establish explicit selection constructs.

b) Concerning the *long-lived* transition, an alternative to clauses naturally comes to mind. It consists in using a conjunction on *sex* and *age* to physically represent the constraint on this transition. The resulting blend is termed a **super-blend** as it is valid for all possible blends resulting from the selections on *sex* and *age*, among which are the actual blends *boy* and *girl*. (To be more precise, the considered super-blend encompasses all possible combinations (x y) where x (resp. y) is a destination of the selection on *sex* (resp. *age*)) Next figure shows the modified graph. Node 15 is the super-blend : it is attached the *long-lived* transition.

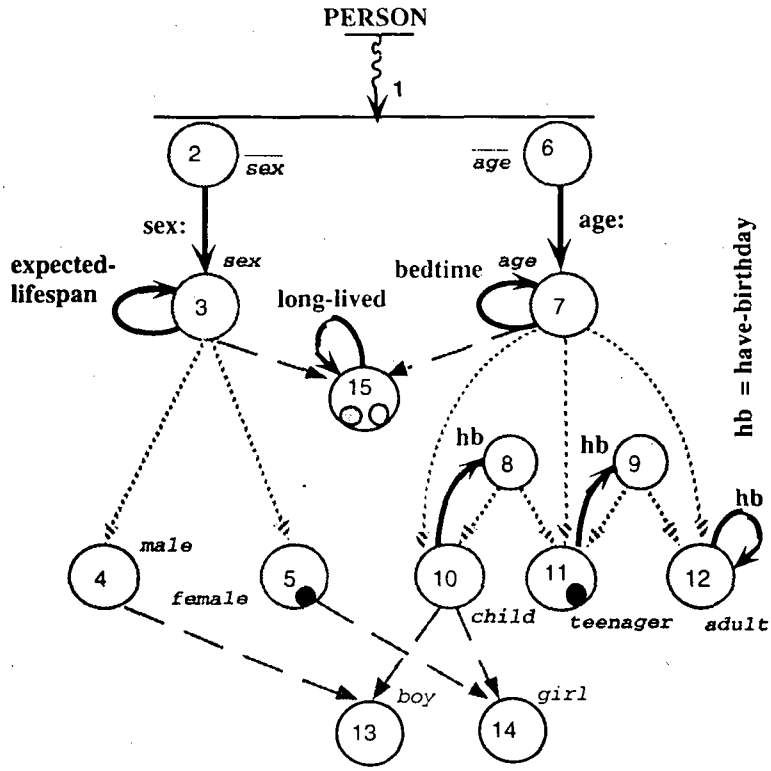


Figure 13.

The existence of this super-blend brings three questions :

-> first question is cognitive : is the modified p-graph easier to understand than the initial one ? The former is slightly more complex than the latter in terms of nodes and reflex transitions ; however, it requires no clause interpretation since *long-lived* is now unconstrained. An ergonomical study will bring the answer. For the moment, we consider the underlying system should support super-blends to let users do what they prefer.

-> second question is about the **"non ubiquity" principle**. This principle requires that an instance state should be represented only once in a color graph : the mark is not to be duplicated. Is the existence of the super-blend playing against this principle ? As a matter of fact, it is not : the mini-tokens marking the super-blend have not the same status than the original mini-tokens. We call these new mini-tokens "traces" (or "replicas").

-> third question is about inheritance rules. The ancestor trees of *boy*, *girl* and other fully initialized states are unchanged, but *long-lived* is no longer inherited in these nodes. The local inheritance rule for transitions needs thus to be modified for enabling a search from the super-blend too. From a conceptual point of view, this can be considered as a new form of multiple inheritance (a form different from the usual structural one).

2) Two important issues

The above variations raise two significant issues :

— first issue, the **cognitive complexity** : depending on what design principle is obeyed or abandoned, a color graph may be more or less complex statically (considering its structure) and/or dynamically (considering its use, notably from an inheritance point of view). Static and dynamic complexities being usually antagonistic, careful experimental studies are necessary to conclude firmly. To quote [Green, 1982], (p. 34), *"The details of the notation profoundly influence its usability (...) A saving too small to be measured [on a small scale] could become a major improvement a thousand times up."* The design principles we adopted represent a bet on the future results of such experiments. Our intent is

to produce a programming environment enabling to swich on or off each design principle (1) for supporting the experiments ; (2) for allowing task-related and/or personal variations ;

— second issue, the **inheritance rules** : the proposed variations —and, more profoundly, the principles sustaining them— do impact the placement of transitions (and methods as well) and hence their retrieving. One important aspect is the resulting form of ancestor trees (actual trees or simple lists ?).

Inheritance is thus one major aspect to take into account for tuning the design principles of color graphs. Are we to decide to restrict LOCAL inheritance to be simple or multiple ? Mono-inheritance is a priori simpler ; multiple inheritance, more powerfull. A study is necessary. The rest of the paper describes it.

Two points will appear :
(a) LOCAL rules for handling trees are actually very simple ;
(b) LOCAL rules can be easily generalized to CLASS inheritance.

Compared to LOCAL simple inheritance, LOCAL multiple inheritance appears more in accordance with CLASS multiple inheritance : we thus propose a support for multiple ancestors in color graphs. This proposal is to be contrasted with the strictly hierarchical style imposed in higraph-based formalisms (unique ancestor).

PART A : INHERITANCE OF TRANSITIONS

This part focuses on the inheritance of TRANSITIONS (implementations will be studied in PART B). First, it describes LOCAL inheritance rules (i.e. inheritance along the reflex transitions of a SINGLE color graph), then it generalizes these rules for handling CLASS inheritance (in a HIERARCHY of classes, i.e. of color graphs).

3. FROM LOCAL TO CLASS INHERITANCE : THE PHILOSOPHY

This section presents an overview of inheritance, from graph-level inheritance to hierarchy-level inheritance. The basic idea and the most fundamental concepts are described.

3.1 THE BASIC IDEA

Having just described the *Person* color graph (figure 7), suppose we are now required to specify the properties of the *Employee* instances. Obviously enough, their color graph will share a lot of informations with the *Person* color graph, even if a *job* dimension will make it different and express new functionalities. Sharing the common parts and making *Employee* derive from *Person* appears most natural, just like in traditional OOP. The difference with traditional OOP is that the information is now structured into color graphs (instead of being flat). Concerning *Employee*, the *job* dimension is expected to remain at the local level. (Alternatively, it can be stored in a dedicated *Job* mixin.)

The basic idea is thus simple : in COP, a class is described in a N-dimensions space. **Each partial description of this class on less dimensions may potentially be itself interpreted as a new class**. This new class is extracted from the initial class. Then, the process may be recursively applied. By construction, the considered class shares a fair amount of informations with the extracted classes. Sharing is thus natural : retrieving the shared information (stored in "superclasses") and constructively merging it with the local remaining information stored in the class itself ("increment") is called class (or hierarchy level) inheritance and combination.

For example, if we consider the *Person* color graph itself, it is clear that the *sex* and *age* dimensions may be separated and captured into a pair of superclasses (see next figure). The *Person* increment describes the *boy* and *girl* blends as well as the *long-lived* transition.

Clearly enough, our "decomposition" operator appears as a pair of scissors for independent dimensions. For mixins, the companion paper n° 2 makes clear that the role of scissors is played by the derivation, a specialized form of decompositon.[13]

Since we already know regular TRANSITIONS are LOCALLY inherited along the reflex transitions[14] ("locally" means "considering a single color graph"), one goal is to build CLASS inheritance for transitions as a generalization of LOCAL inheritance. (Concerning the IMPLEMENTATION aspects, the idea is to hypothesize that methods or memory representations may also be inherited along reflex transitions (LOCAL inheritance), with a dependence on dimensions, and then to generalize to CLASS inheritance.)

[13] This is an intuitive point of view, since the decomposition and derivation are used for gluing behaviours : in fact, their existence makes the separation of parts possible.

[14] except those flowing from a decomposition....

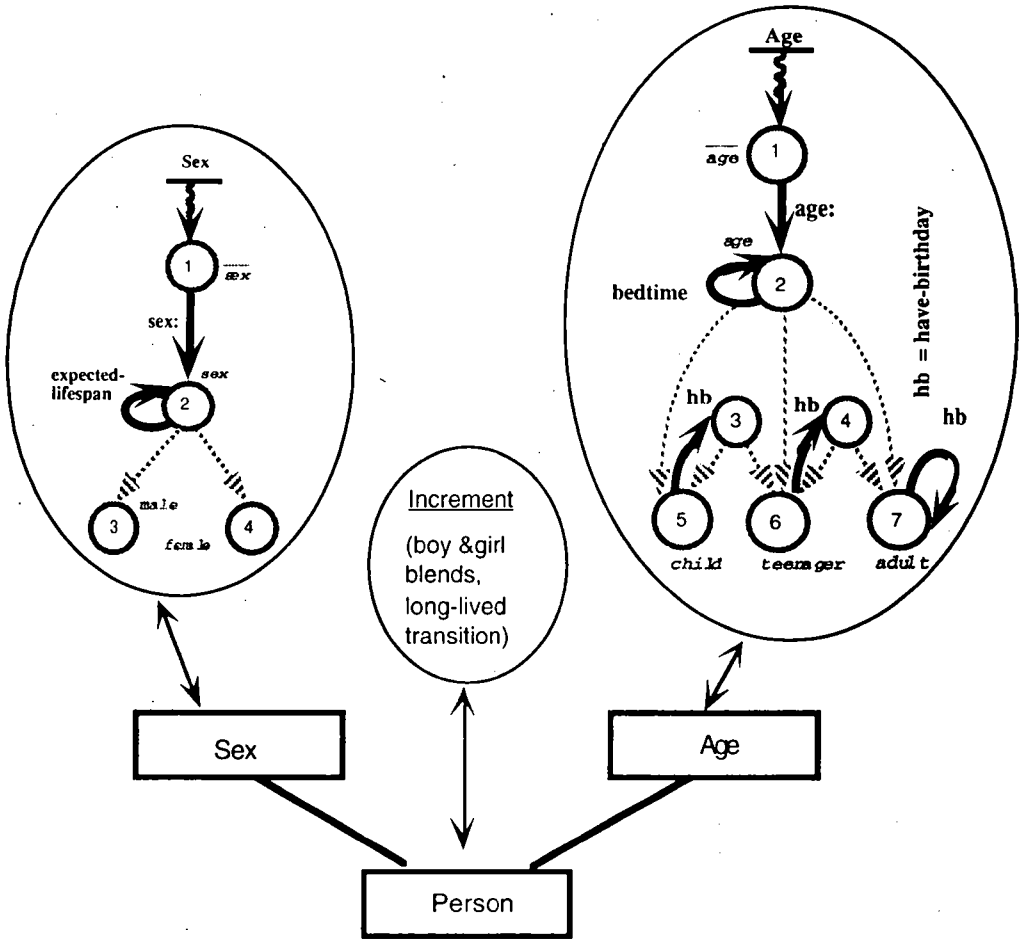


Figure 14.

3.2 KEY CONCEPTS

Key points in the proposed description of inheritance are the concept of dimensions, inheritance along reflex transitions and ancestor trees. These points have already been presented in companion paper n° 1. Yet, to make this paper self-contained, we examine these points again, possibly in more details.

3.2.1 The concept of dimension

A color graph is a graphical representation of a system of (discrete) states and transitions.[15] From a theoretical point of view, the representation to be considered as a reference is a cartesian coordinate system in a N-dimensions space. In such a space, each dimension is materialized by one axis. Each state is described as a point in this space, i.e. by N coordinate points a priori, its projections onto each axis (a coordinate point is termed a color (N=1) or a pigment (N>1)).

Next figure illustrates that with a *Circle* skeleton. Two dimensions are used : the *radius* and *center* dimensions. Along the *radius* dimension, two points (pigments) exist : one corresponds to *radius-uninitialized* (*not r*) ; one to *radius-initialized* (*r*). Idem along the *center* dimension (*not c* ; *c*). These two pairs of two pigments along each orthogonal axis determine four points, the four reachable states of a *Circle*. For instance, a fully initialized instance (state 4) is represented by the couple (*r c*)/

[15] An alternate representation, grounded on insideness instead of connectedness, is one that derives from higraphs.

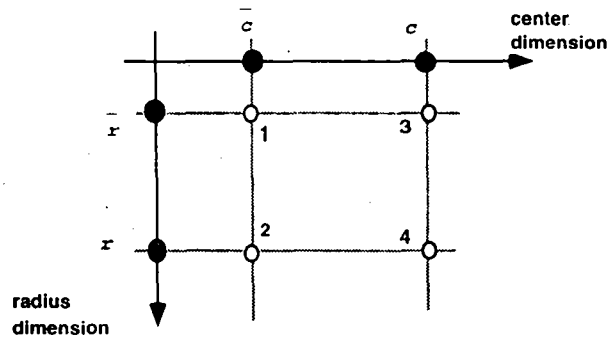


Figure 15.

Like the cartesian space, the color graph formalism enables the (easy) representation of points as such (ex. : point 4), and —more generally— of sets of points (subspaces) that are parallel to one or several axes (ex. : a line in 3-dimensions space). [16] This is what we name blends (subspaces), among which are the colors (true points). For example, point 4 of the above figure can be shown in a color graph as a blend (acually, it happens to be also a color).

Any transition between two states can itself be decomposed into its projections onto the coordinate axes. For example, the *radius:* transition can be applied to point 1 or to point 3. It is represented as a simple, unconstrained transition along the *radius* axis (from *not r* to *r*). The *draw* transition can also be represented by its projections onto the axes : this leads to a coupleof circular constrained transitions respectively attached to *r* and *c*. Given our *Circle* example, the *draw* transition is in fact more simply represented when directly attached to the point 4. (See next figure.)

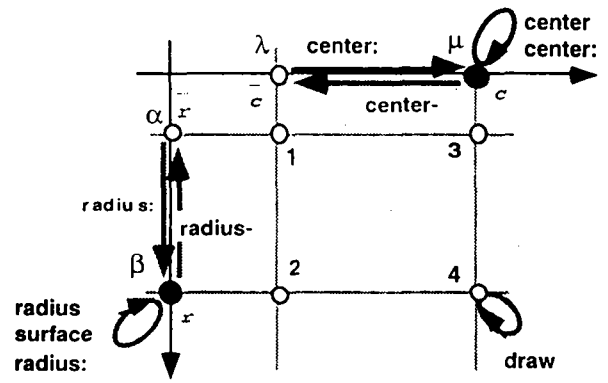


Figure 16.

Here is the corresponding color graph.

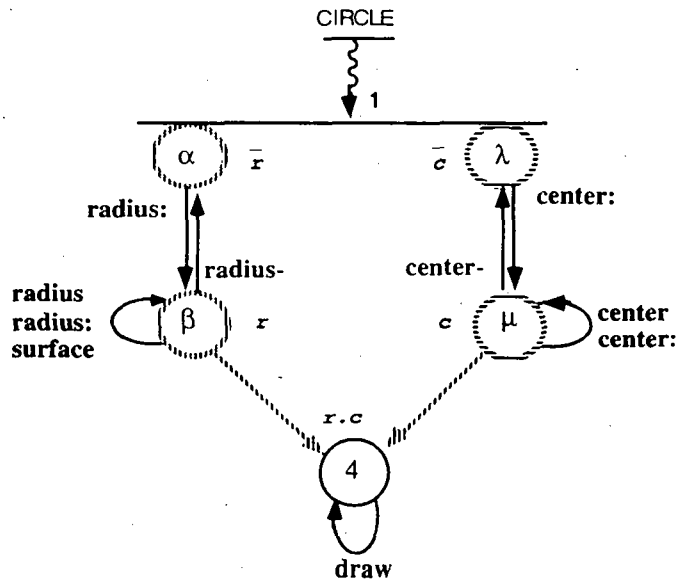


Figure 17.

[16] This statement is not valid when using a formalism deriving from higraphs.

The cartesian representation is useful for interpreting the decomposition of states and transitions according to N-dimensions. However, it is rarely practical : human beings having difficulties to vizualize a cartesian space when N exceeds 3. Hence, the color graph representation.

3.2.2 Inheritance along reflex transitions

The fundamental LOCAL INHERITANCE RULE FOR TRANSITIONS was established in companion paper n°1 and re-stated extremely briefly in §2.2.2. This subsection examines it in details and shows it can be enlarged to implementations.

• If we examine the **conjunction** in the above figure, we notice that the transitions flowing out of pigments β and μ (resp. *surface*, *radius*, *radius:* and *radius-* ; *center*, *center:* and *center-*) are valid in node 4. (This is because the condition of this node, a blend, ANDs the condition of pigments β and μ.) All happens as if the origin of the transitions in question could slip along the reflex transitions (β 4) and (μ 4).
This statement can be enlarged to methods and memory representations as well. For example, the *surface* method, attached to node β, will be valid (inherited) in node 4 ; the *radius* and *center* cells (slots), respectively defined in β and μ, will both be present (combined) in 4.

• Reflex transitions also appear in **selections**. So, let's examine if inheritance works also in such constructs. Next figure redraws the *Age* class shown in figure 14 while positioning all the basic colors (6, 10, 11, 12) along the *age* axis. Here, the ephemere colors (7, 8, 9) are not represented on the axis, but above it along some imaginary dimension[17]. In addition to materialize the fact that tests needs to be done, these nodes may also be used to factorize transitions (ex. : *bedtime* in 7) as well as memory representations (ex. : an *age* cell in 7) and methods (ex. : *have-birthday* in 7). Here again, transitions, methods and memory representations slip (are inherited) along the reflex transitions[18].

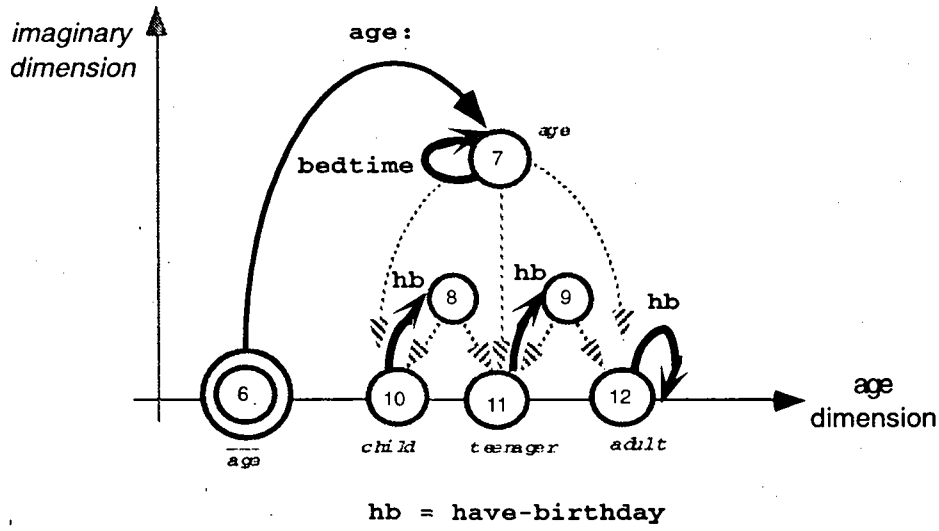


Figure 18.

Inheritance rules in selections somehow express that ephemere colors are —strictly speaking— not necessary. Next figure shows how the preceding one gets transformed when ephemere nodes are banned.

---

17 Other representations are possible, although less expressive : an ephemere color recursively corresponds to a cloud of basic colors ; it could also be shown as an extra point on the same axis as a bsic color.

18 The reader of the second companion paper will have noticed the similarity between this representation and the canonic skeleton a mixin is given. The corresponding c-graph may have been given the same structure, as well as the other c- or p-graphs presented in this paper.

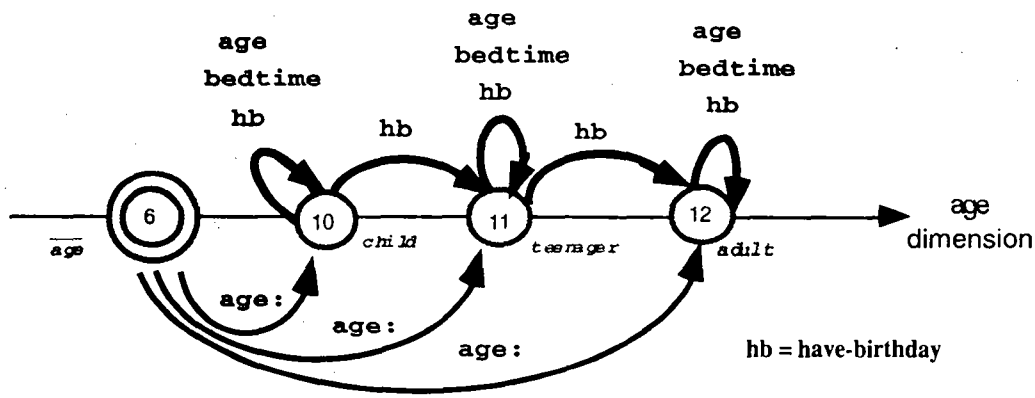


Figure 19.

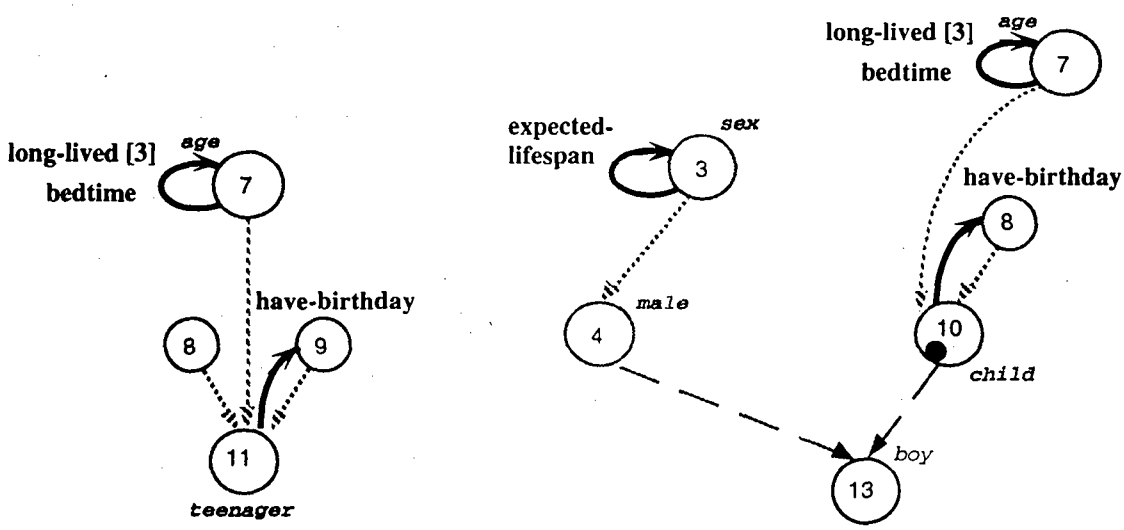
• Finally, reflex transitions also conceptually appear in **decomposition** constructs. However, they are not inherited in this case.. (Our visual representation reflects that : a decomposition construct is drawn as a bar, and not as a diverging tree of reflex transitions.)

3.2.3 The concept of ancestor tree

Regular transitions (as well as methods and memory representations) being inherited along reflex transitions, the concept of ancestor tree (see §2.2.1) emerges immediately as an important one when interested in gathering inherited properties in a given node.

More precise definitions are used :
— in a c-graph, the ancestor tree of a node is also termed its **c-ancestor-tree** ;
— in a p-graph, the ancestor tree gets a priori more complex as it depends on more involved dimensions : it may be **simple** (if the root is a pigment) or **multiple** (if the root is a blend of degree d). We term **p-ancestor-tree** the restriction of the ancestor tree to the nodes participating to one given dimension (i.e. which may hold the dedicated mini-token).

Let's take two examples in the *Person* color graph (figure 7). The ancestor tree of *teenager* (pigment 11) is simple : it is a single p-ancestor-tree made by the reflex transitions (7 11), (8 11), (9 11) along the *age* dimension (figure 20). The ancestor tree of *boy* (blend 13) is multiple. Its p-ancestor-trees are (figure 21) :
- for the *sex* dimension, the reflex transitions (3 4), (4 13) ;
- for the *age* dimension, the reflex transitions (7 10), (8 10), (10 13).



Figures 20 & 21.

4. LOCAL INHERITANCE OF TRANSITIONS (IN ONE COLOR GRAPH)

4.1 IN A C-GRAPH

4.1.1 Preamble

A c-graph is a color graph that exhibits but one dimension (N=1). The *Age* c-graph of figure 14, redrawn in figure 18, typically depicts such a case. Except for the pseudo-color, any node is a color : it corresponds either to one reachable instance state (it's a basic color) or to a set of reachable instance states (it's an ephemere color).

An ephemere color is the source of a selection, the only construct that may be instantiated in a c-graph. Two or more selections may share a same destination color, thus creating a **pseudo-conjunction** : this destination color has several fathers. Yet, this is not the common case : usually, a a color participates to at most one selection. The ancestor-tree (c-ancestor-tree) of any color is thus a list (cf. the *Sex* c-graph in figure 14) or, more rarely, a tree (cf. the *Age* c-graph in figure 14).

As already established by the LOCAL INHERITANCE RULE FOR TRANSITIONS (see companion paper n°1), regular (graph) transitions are inherited along the reflex transitions of a c-graph.

4.1.2 Inheritance of transitions in a c-graph

a) Precise semantics

"Inheritance along reflex transitions" should be understood in the following way : if a graph transition *T* exists in the c-ancestor-tree of a basic color, a graph transition similar to *T* conceptually flows from this basic color ... to the destination of *T*. This transition is termed the "inherited transition".

This statement has two aspects :

1) a source aspect :
the considered color is source of an inherited graph transition *T*. Since colors directly correspond to states or to sets of states, any graph transition between two colors corresponds to a state transition or to a set of state transitions. Thus, any instance in this basic color (i.e. in the stable state materialized by this basic color) may validly be sent the message *T* . (This validity in term of message immediately extends to generic functions, i.e. when several color graph instances are involved) ;

2) a destination aspect :
the destination of the *T* graph transition defines the new instance state once *T* has been received. Generally speaking, this destination is to be determined at run-time among the ultimate **descendants** of the destination of *T* (i.e. among the leaves of the tree of reflex transitions recursively flowing from this destination node). Of course, if this tree is reduced to a single (basic) color, then the destination of *T* is that color itself and no run-time testing is necessary. Except this trivial case, the destination of *T* may also be determined statically when *T* is an i-circular transition : the inherited graph transition is also i-circular, thus its destination is its source (the considered basic color).
Companion paper n° 1 states these results using a token to mark the current state. Two steps are distinguished. **Move step** : the token stays in its position if the transition is i-circular (final position) ; otherwise, the token is moved to the destination of the transition. **Propagation step** : reflex transitions are activated : a number of them (selections) may then fire, hence the new instance state. [This algorithm presupposes the c-graph satisfies the "unique destination" principle[19].]

b) Examples

Let's consider, for instance, the *Age* c-graph of figure 14. The transition labelled *betdime* is i-circular : it is obviously inherited in basic colors 10, 11 and 12, each time as a circular transition (static property). Suppose now the *have-birthday* transition in the *Age* c-graph has been declared as a g-circular transition attached to the ephemere color 7. Then, it would be inherited in 10 (*child*). If activated in this color, then may have been activated for a *child* instance, moves the token from 10 to 8 ; a test is then done ; depending on its result, the token is then moved to 10 (*child*) or 11 (*teenager*).

---

[19] If the "unique destination" principle is disobeyed, several (inherited and/or locally defined) similar transitions are elected for firing. The corresponding set of destinations is collected and tests are done (as if an ephemere node was existing). From an animation point of view, the token may be shown as visiting, for each transition in question, its source and then its destination. This may be done either in sequence (dynamic view) or "in parallel" (static view). Whatever the animation, when tests are done, a unique destination is selected : the token gets in there (if others were shown in the animation (static view), they disappear).

4.1.3 At most a single transition T is ever valid in one color

As expressed above, the ancestor-tree of a color is either a list or a tree. The first case potentially corresponds to LOCAL SIMPLE inheritance ; the second case, to LOCAL MULTIPLE inheritance. As a matter of fact, multiplicity does not arise if the "unique destination" principle is enforced : no two similar transitions (each one being locally defined or inherited) may be valid at the same time in a given color. (Figure 20 shows an example.) Reference [Borron, 1996b] discusses variations and facilities. Yet strictly not necessary in a c-graph, a facility can be adopted : the **masking** by a local transition of inherited similar transitions. (If this facility is used, then there is still one valid transition in a given color.)

4.2 IN A P-GRAPH

4.2.1 Preamble

A p-graph is a color graph that exhibits two or more dimensions (N>1). Whereas a c-graph describes states as points along one same axis, a p-graph decomposes a state onto several axes (one axis per dimension). As in a c-graph, we may also add an imaginary dimension for selections. Next figure examplifies this addition, showing the *Person* p-graph of figure 7 in 3 dimensions (N+1).

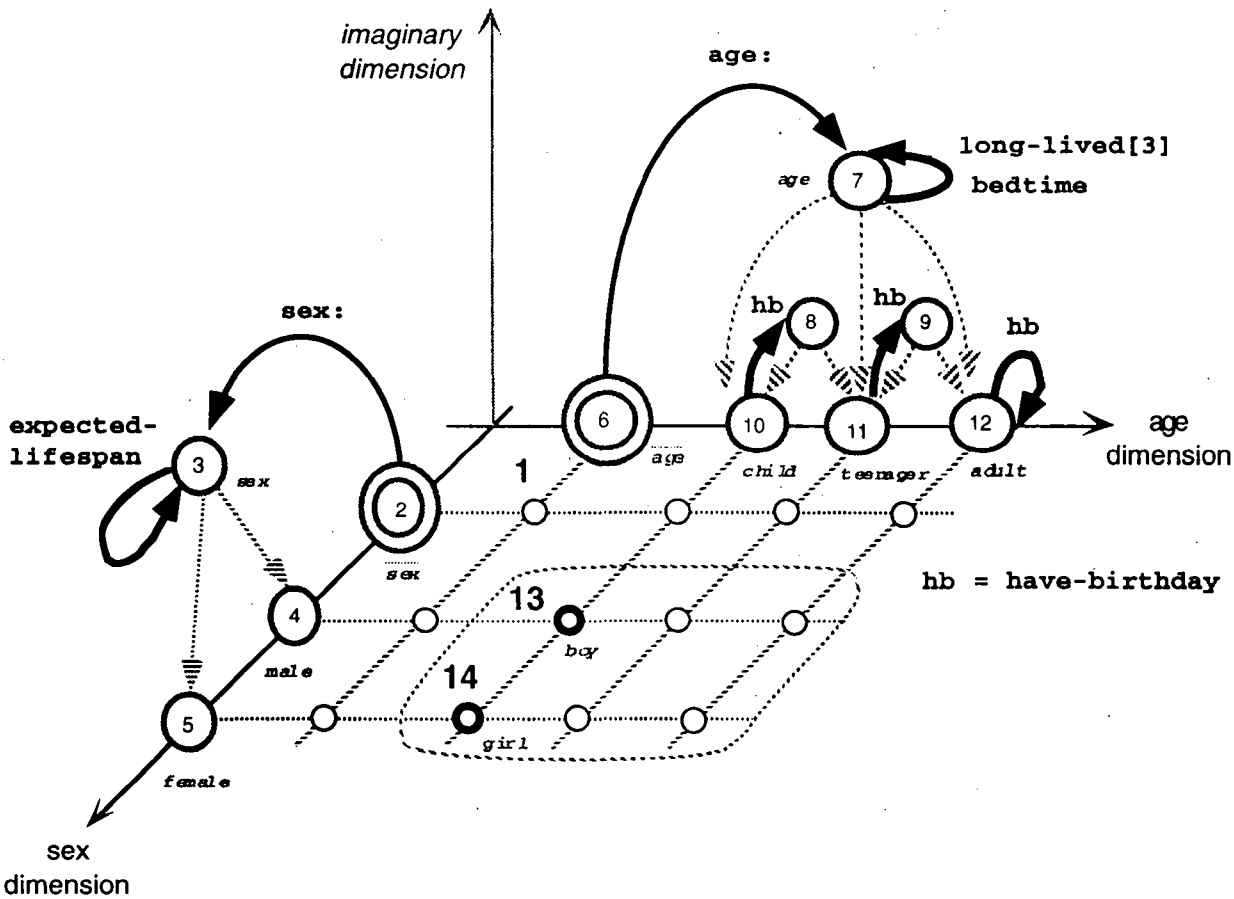


Figure 22.

• Let's comment a bit the figure. In a p-graph, any node (except a pseudo-color) is a termed a p-chroma. As in a c-graph, a distinction is made between a basic and an ephemere p-chroma. A basic p-chroma belongs to the space S strictly defined by the N-dimensions ; an ephemere p-chroma, to the complement of S in the space defined by the N-dimensions plus the imaginary one.

A basic p-chroma is either a basic pigment (like 2, 4, 5, 6, 10, 11 and 12) or a basic blend (like 13 and 14). The first category describes an atomic substate along one single dimension ; the second one, a composite substate along several dimensions, possibly N —in which case the blend is also termed a color.

An ephemere p-chroma is either an ephemere pigment (like 3, 7, 8 and 9) or an ephemere blend (none is shown in the figure, but *long-lived* could well be implemented using one[20]).

• Because it describes a substate, a p-chroma is usually not associated to a single instance state but to a set of instance states (all those that match the associated substate). Conversely, a reachable instance state is usually not represented by a single node (as it is the case in a c-graph) but by k p-chromas, the degrees of which sum to N.

• In a p-graph, three constructs may be instantiated, the selection (as in a c-graph), the decomposition and the conjunction. As stated by the LOCAL INHERITANCE RULE FOR TRANSITIONS, regular (graph) transitions are inherited along the reflex transitions of a p-graph (except when originating from a decomposition).

4.2.2 Inheritance of transitions in a p-graph

a) Relationship with inheritance in a c-graph

Let's establish the relationship with the preceding analysis about inheritance in a c-graph.

1) **the ancestor-tree of a pigment** (in a p-graph) **is exactly like the c-ancestor-tree of a color** (in a c-graph) considering the restriction of the p-graph to the dimension in question. Such a restriction is termed a **p-ancestor-tree.** A graph transition in the p-ancestor-tree of a pigment is thus inherited just like a graph transition in a c-graph : results previously established apply.

   As an example, one can consider the *age* dimension of a*Person* instance (figure 22) : the restriction in question exactly coincides with figure 18, the c-graph of *Age* : this proves the result for each pigment of the *age* dimension of *Person,* in particular for pigment *teenager* (see figure 20) ;

2) **the ancestor-tree of a blend of degree D is interpreted as the superposition of D p-ancestor-trees** (one per dimension involved in the blend). For example, the blend *boy*, being a (*male child*) inherits both from pigments 4 (*male*) and 3 (*sex*), and from pigments 10 (*child*), 8 and 7 (*age*) (see figure 21).

   Let's add two remarks :

   -> Two p-ancestor-trees of a same blend may share a reflex transition or a list of reflex transitions : this occurs when the blend in question recursively results from a conjuction involving at least one source blend (this requires N≥3). The definition of a p-ancestor-tree takes care of that. In this respect, the given example is particular since *boy* is directly made from pigments : the *age* and *sex* p-ancestor-trees are completely distinct.

   -> If ephemere blends are used (which is not recommended because of the complexity generally induced), the inheritance structure they constitute extends only over blends of same degree and thus does not interfere with the inheritance structures that extend over pigments. All these structures are lists or trees, and not DAGs nor more general graphs, an important point for keeping inheritance easy to deal with. In our example, no ephemere blend is built on *boy* and and other blends. Yet, it could be : in particular, the*long-lived* transition may be implemented that way as already noted.

---

[20] The *long-lived* transition, which is constrained both by node 3 (its clause) and node 7 (its source), could well be attached to an ephemere blend, the unique role of which would be to factorize *long-lived* in one place. However, this would require the six blends in question (colors) to be displayed -instead of two presently, which is uneconomical. (A cheaper solution consists in using a super-blend as done in figure 13.)

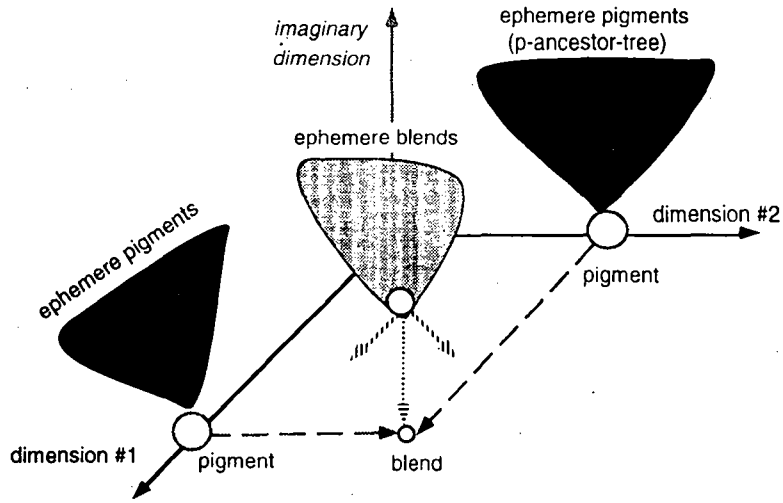


Figure 23.

Given these two properties, the basic schema to deal with inheritance in a p-graph is the following :
- for each involved dimension, activate the inheritance scheme used for a c-graph ;
- then combine the obtained results, i.e. the destinations of the activated transitions.

The determination of the effective destination state upon the reception of a message is thus a bit more complex than in a c-graph, especially when a composite transition has been activated. It is better explained in terms of mini-tokens (see companion paper n°1). Besides the move and propagation steps defined for a c-graph, a backtracking steps exists. **Move step** : for each non i-circular triggered (micro-)transition, the involved mini-tokens are moved to the transition destination. **Backtracking step** : moves may have depleted a number of blends ; the mini-tokens that remain in them are moved back to their original pigments. **Propagation step** : reflex transitions are activated : a number of them (selections and conjunctions) may fire, hence the new instance state. [This algorithm presupposes that the p-graph satisfies the "unique destination" principle vs. each dimension. As for a c-graph, masking may be used.]

b) New features to be taken care of.

b.1) transition validity

In a p-graph, the transitions inherited in a given p-chroma are not systematically valid (as they are in a c-graph) : they may be constrained by a clause the effect of which is to reduce the validity of the transition to a subset of the set of instance states associated to this p-chroma.

For example, in *Person*, the transition *long-lived*, attached to the pigment 7 (*age* initialized), is constrained by clause *[3]* (*sex* initialized) : instead of nine substates, only six are concerned by *long-lived* : the ones encircled by a dashed line in figure 22.

Given a reachable instance state, we consider the k p-chromas materializing this state and the transitions that are inherited in these p-chromas :
-> if unconstrained, an inherited transition is valid ;
-> if constrained, an inherited transition (be it simple or part of a composite transition) is valid if the instance state satisfies its clause[21].

b.2) multiple transitions

• The algorithm given above presupposes a unique graph transition (at most[22]) is found per dimension.

-> Concerning pigments, the "unique destination principle" is normally enforced in their p-ancestor-trees as it is in a c-graph. Thus, given remark a.1, no two similar transitions may be valid at the same time in a given pigment.

---

[21] Note that the elements of a composite transition are all constrained in the same way, the compositions of their respective pairs of clauses and source p-chromas being all equal.

[22] Those that are not found are circular (see companion paper n°1).

-> Concerning blends, the principle cannot be systematically obeyed unless ephemere blends are taken advantage of. Since these nodes generally add complexity to a color graph[23], they happen to be rarely used. Equivalent modellings are chosen in place :
— a general solution is offered by multiple composite transitions : yet, the visual simplicity of the p-graph is an apparence ; it is counter-balanced by the complexity of clauses ;
— to offer, in certain circumstances, a cognitively simpler expression, transitions to "super-blends" (cf. §2.4.2.1.b) and "negative" transitions have been invented. They are discussed in [Borron, 1996b].

• In case of multiple composite transitions, the above algorithm is upgraded as already done in case of multiple transitions in a c-graph. The extension is based on the idea that creating an ephemere node would solve the problem. The algorithm thus proceeds as if we were going to build it (compute all the possible destinations states) and to use it (test the conditions[24]). In fact, the next states need not be computed as such : only the involved dimensions are taken into account for determining the necessary tests. Once a composite transition has been finally chosen, the three steps of the above algorithm are gone thru.

Next figure illustrates such a situation. It presents a case where a T transition changes the instance state from [α β] to either (α β′), (α′ β) or (α′ β′). Using an ephemere blend leads to a relatively complex p-graph. The figure uses three composite transitions in place.

Applying the upgraded algorithm, the transitions destinations are first collected : (α β′), (α′ β), (α′ β′) ; they are tested and one is chosen ; then the two mini-tokens are moved accordingly.

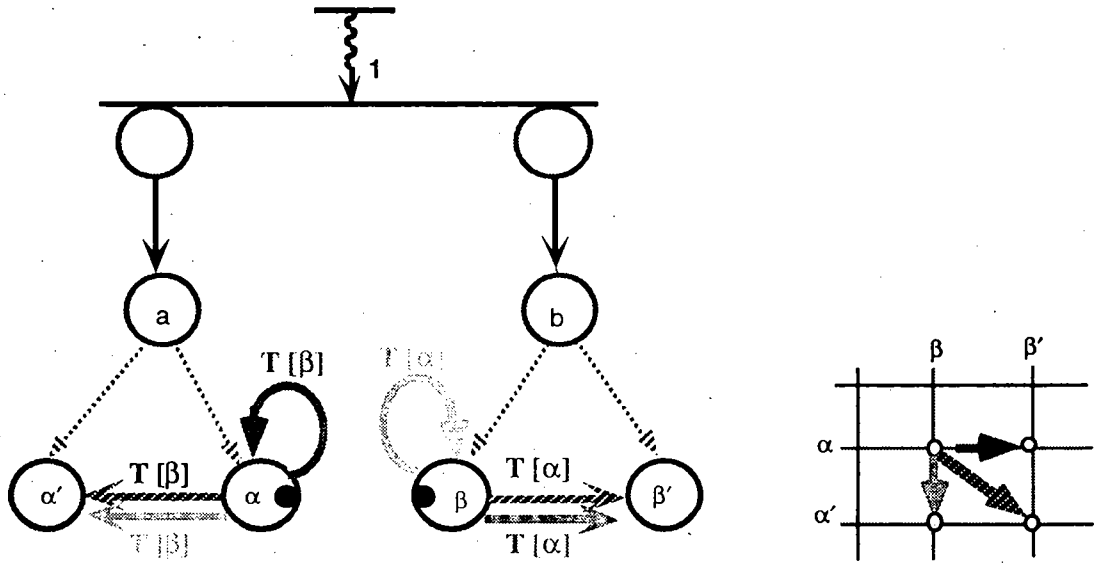


Figure 24.

As noted above, a modelling with multiple composite transitions (equivalent to what can be found in a higraph) is not fully satisfactory from a cognitive point of view. In [Borron, 1996b], a different modelling is thus proposed as an alternative for this same example.

23 They require all destination blends to appear (risk of combinatorial explosion) and the structure originating from them is not flowing downwards as selections and conjunctions do (in figure 7, for ex.).

24 IMPORTANT : this supposes the code to be run before the evaluation of conditions is the same in each case.

5. CLASS INHERITANCE OF TRANSITIONS (IN A HIERARCHY OF COLOR GRAPHS)

Next subsection explains the relationship between inheritance of transitions in a hierarchy of color graphs and inheritance of transitions in a p-graph. The subsection afterwards introduces operators for putting class inheritance in practice.

5.1 RELATIONSHIP WITH INHERITANCE IN A P-GRAPH

The relationship is first shown informally using the *Person* example ; basic class inheritances rules are formulated ; main consequences are apprehended ; then the *Person* example is formalized as an introduction to operators and naming facilities.

5.1.1 Relationship

a) Motivation

Looking at the *Person* p-graph given in figure 7 or, equivalently, at its cartesian representation in figure 22, the quasi-independence of the *age* and *sex* dimensions appears as an invitation to split this p-graph into several parts. Let's term **p-subgraph** vs. a given scale (dimension) the restriction of a p-graph to pigments of one given scale : in the *Person* p-graph, we can distinguish the *age* and *sex* p-subgraphs. Hence, three parts :

— the first and second parts are devised for separately holding the behaviour materialized by the *age* and *sex* p-subgraphs. These parts are c-graphs (see figure 14) and each one naturally corresponds to a class (say, *Age* and *Sex*) : no new mechanism is necessary for describing them ;

— a third part is meant for capturing all what requires both dimensions (the *long-lived* transition, the *boy* and *girl* blends). This imposes the description of an increment.

Given this splitting, the *Person* p-graph is obtained by the composition of the two extracted classes (now termed **superclasses**) and by the addition of the increment.

b) Modelling

Next figure models the situation :

(1) the *Person* class is attached an incomplete p-graph (the full skeleton of *Person* together with the *long-lived* transition) and knows *Sex* and *Age* as its superclasses ;

(2) the *Sex* class is attached the complete *Sex* c-graph (skeleton + transitions) ;

(3) the *Age* class is attached the complete *Age* c-graph (skeleton + transitions).

[In the figure, the *Person* skeleton is shown as a multi-level selection : this is to preserve its connexity (it lacks the *age:* and *sex:* transitions) ; the *Age* and *Sex* c-graphs are represented in the same way, this to keep the resemblance with the *Person* skeleton ; finally, ephemere nodes having a t(rue condition are visually simplified (no bubble). An alternative would be to represent the superclasses c-graphs normally and the *Person* increment like the *Person* p-graph yet without any labels except for *boy*, *girl* and *long-lived*.]

The instance current state is supposed to be *boy*.

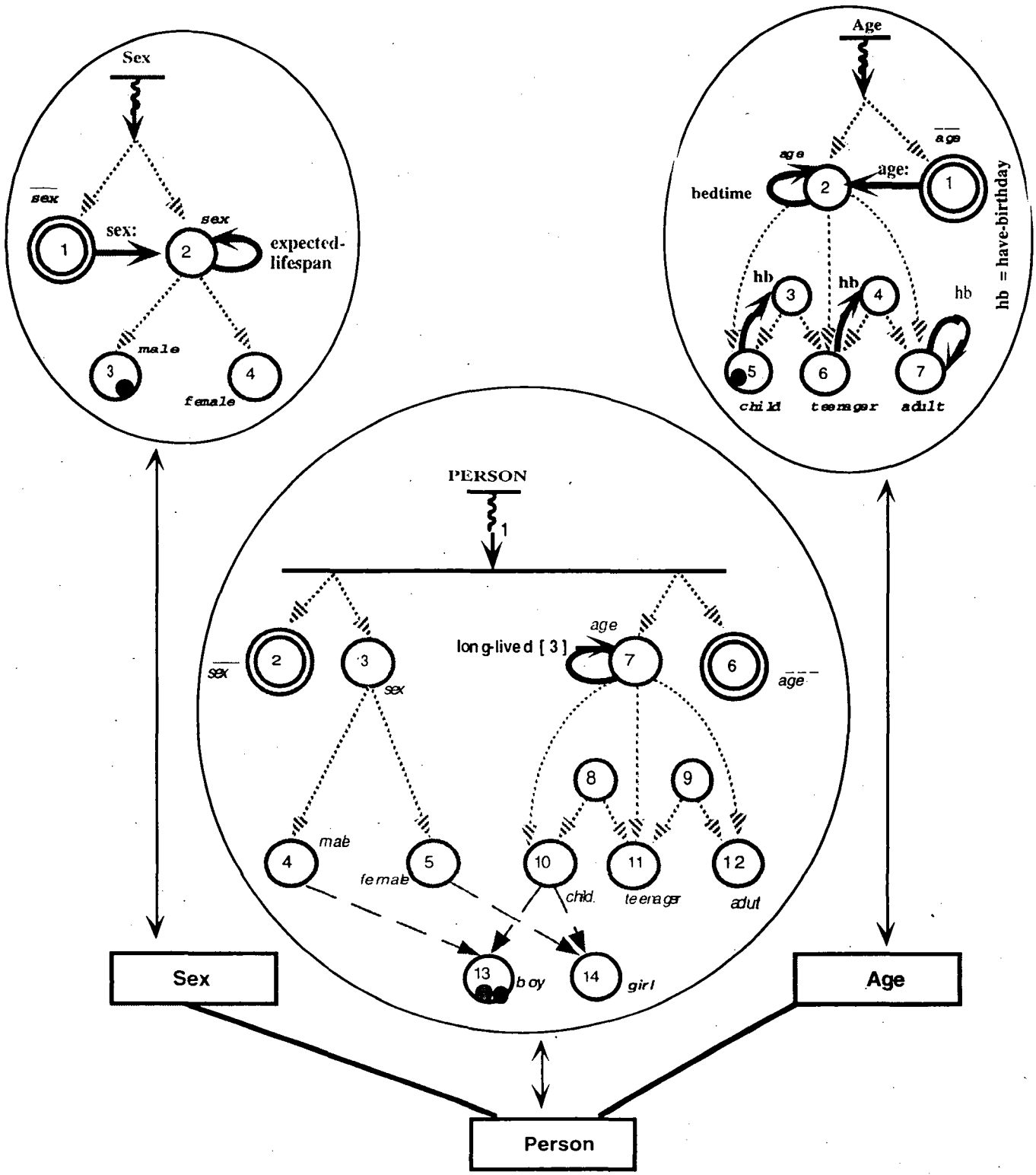


Figure 25.

Next figure models the situation using a cartesian representation. Note the clear distinction between, for example, the *age* dimension and the *Age* color graph.

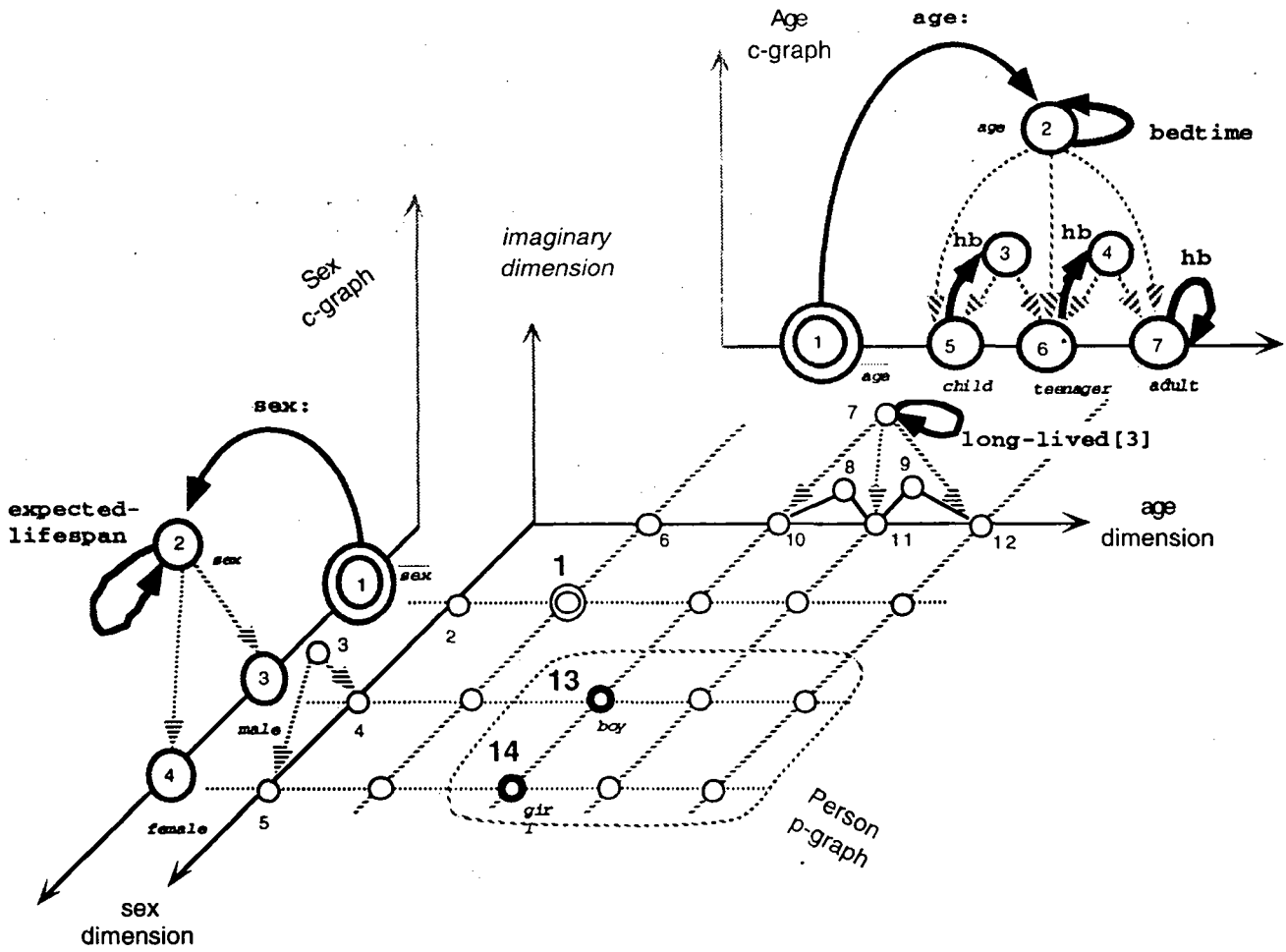


Figure 26.

Note that the inheritance link between a superclass and its subclass, unique in figure 25, is now duplicated times the number of colors present in the considered superclass c-graph. For example, four colors exist in the *Sex* c-graph ; and four inheritance links exist between this graph and the *sex* p-subgraph. Yet not all represented in the above figure for simplicity purpose, these are —mentionning first the class pigment id and then the superclass pigment id : the (5 4), (4 3), (3 2) and (2 1) links. In each case, the pigment and the color of a same couple share the same condition, for example, *male* for the couple (4 3).

c) The composition operator

How to obtain the desired effect ? As a matter of fact, the **decomposition** construct does almost what is needed : instead of specifying the complete *age* and *sex* p-subgraphs, what we have to do is to name the *Age* and *Sex* superclasses so as to get and compose their respective color graphs ; concerning the increment, we have to correctly attach the *long-lived* transition and to correctly compose the *boy* and *girl* blends (and accompanying conditions/transitions) : we thus need to be able to unambiguously name a node in a superclass. Hence, we basically need : (1) an operator (let's term it the **composition**) ; (2) a naming mechanism.

Subsection 5.1.4 illustrates this more precisely. For the moment, let's show the visual representation we chose (next figure) since it makes apparent the relationship between the **composition** operator and the underlying **decomposition** construct : the horizontal bar, which usually denotes a **decomposition**, is here surmounted with the name of each superclass. The instance current state is supposed to be *boy*.

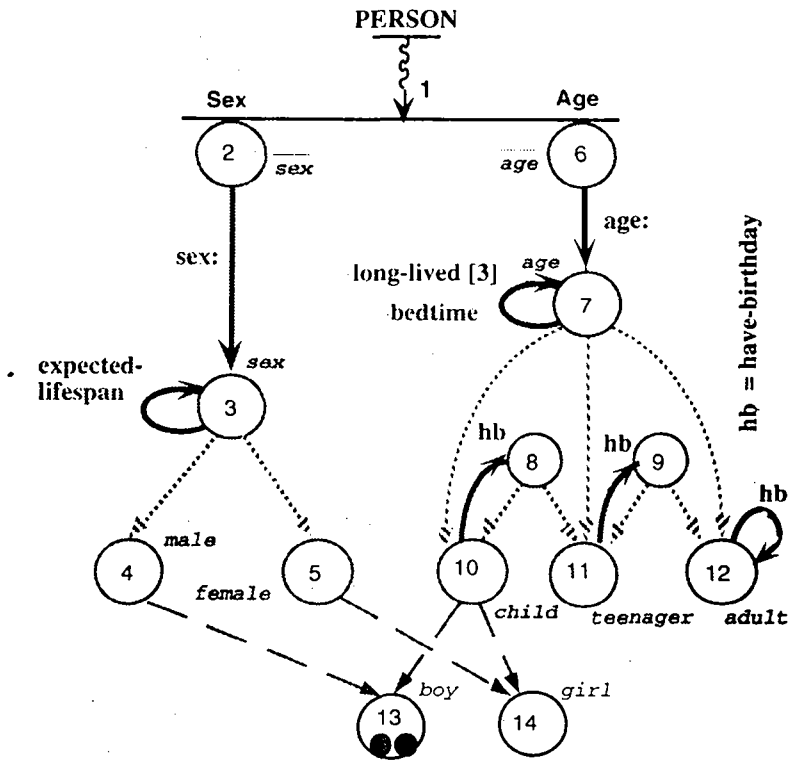


Figure 27.

Next figure abstracts a bit the preceding one : it essentially exhibits the increment and its relationship with the two superclasses. *Long-lived* has been attached to the super-blend (3 7) to emphasize that this transition requires both dimensions.

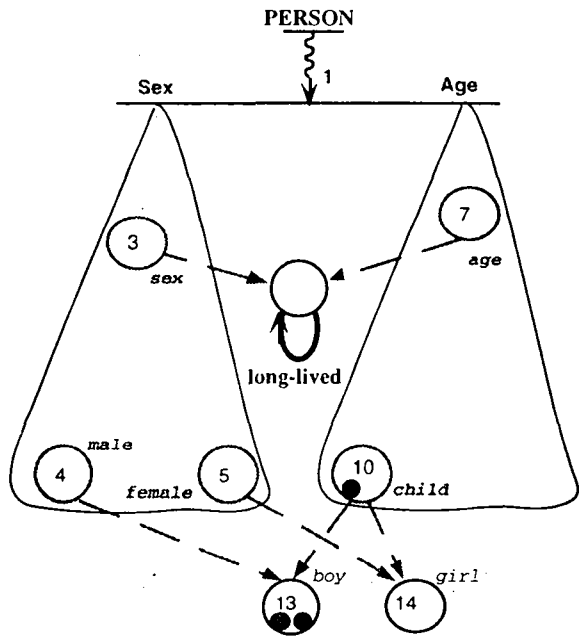


Figure 28.

d) Inheritance

From the hierarchy specifier's point of view, *Person* inherits from *Sex* and *Age*. When a *Person* instance in a given state is sent a message, the validity of this message and its effect are computed using the local increment as well as the *Sex* and *Age* c-graphs.

The instance state is represented by two mini-tokens in the *Person* skeleton, by one token in the *Sex* c-graph and one token in the *Age* c-graph (cf. figure 25). Moves of the mini-tokens are constrained by those of the tokens.

Let's consider a *Person* instance and let's suppose its current state is *boy*.
When receiving a message like *expected-lifespan*, the search first starts locally (i.e. in the *Person* skeleton) according to the LOCAL inheritance rules previously enumerated for p-graphs, then the search is extended to the ancestor graphs, here the *Sex* c-graph and the *Age* c-graph —in this order. LOCAL inheritance rules are used inside each ancestor graph. The search is made recursively : if the *Sex* and *Age* graphs were themselves obtained by composing ancestor graphs, then the search would also be done in these ancestor graphs. Here, a single *expected-lifespan* transition is found in the *Sex* c-graph. The instance state is unchanged : the *Sex* token is not moved (*expected-lifespan* is i-circular) ; the *Age* token, too (it is not involved) ; the *Person* mini-tokens thus stay in place.
Suppose the next message is *have-birthday*. New searches are done in the same manner ; a *have-birthday* transition is thus found in the *Age* color graph, possibly making the *Age* token move to *teenager* ; the *age* mini-token in *Person* is moved accordingly ; the *sex* mini-token bactracks to *male* ; hence, the new instance state : (*male*, *teenager*).

All happens as if the inherited transitions were locally attached : finding the *have-birthday* (resp. *expected-lifespan*) transition in the*Age* (resp. *Sex*) c-graph is like finding it in the *age* (resp. *sex*) p-subgraph of the *Person* p-graph. The inherited transitions can be conceptually attached to the p-graph : in this case, the two tokens in *Sex* and *Age* get useless. Figure 25 is better in accordance with this point of view.

5.1.2 Basic class inheritance rules

As expressed above, the composed p-graph **inherits** transitions from its ancestors : rules are needed to predict the global behaviour solely from the color graph pieces (superclasses + increment). ***This is obtained by extending the local inheritance rules used in a p-graph.*** Two slightly different problems are encountered : computing all possible transitions in a given instance state (i.e. computing the transition dictionary for this state) ; getting the transition -if it exists- that is similar to a message in a given state.

Given a instance state, the transitions that are valid in this state are searched dimension per dimension, first locally (see the inheritance rules in a p-graph), then in the color graphs attached to the superclasses of the instance class (in the order defined by the list of superclasses).
The algorithm is recursive. (Originally, it is broad first : a class is computed once all its superclasses have been computed ; however, it can can be made depth first with some care : corresponding details will be elaborated subsequently.) If masking is allowed, a successful search prevents recursion.
The obtained transitions can be thought as being locally attached to the considered p-graph : in this case, the transition destinations are obtained exactly like in a regular p-graph. (Alternatively, one may prefer to consider the transitions to be still attached to the ancestor color graphs : in this case, before the backtracking and the propagation steps, the moves of the mini-tokens in the composed p-graph are driven by the moves of marks in the ancestor graphs.)

This algorithm is easily modified for getting but the transition -if it exists- that is similar to a message in a given state.

5.1.3 Important consequences

a) Monolithic and hierarchical views

Two mechanisms now exist for a class to be given a color graph : one is simply to create this color graph from scratch (**defcolorgraph**) ; one is to build it as a (recursive) **composition** of the color graphs of the ancestors of this class. In any case, the resulting color graph (termed the "**global color graph**") monolithically describes the interface of the class, i.e. a specific behaviour (supposed to be stable). This global color graph is invariant vs. its specification in terms of a hierarchy.

The monolithic and the hierarchical views have their own merits.
-> The user of a class is interested in the first one, i.e. in the global color graph as such and not in the way it is possibly composed. For this user, inheritance is local (internal to one color graph).
-> On the opposite, the specifier of the global color graph is interested in the hierarchical view : he/she aims at providing the corresponding behaviour by assembling scattered color graph pieces. For this specifier, **class inheritance** is essential : his/her point of view is very similar to the one of a traditional OOP class implementor.

b) Precision in inheritance

As already noted, a class is described in a N-dimensions space in COP and as a single point in traditional OOP. Hence, one major difference clearly appears between these two forms of programming : inheritance is much more precise in COP than in OOP. This is like having micro-surgery instruments compared to domestic ones : the result is thiner and obtained more elegantly.

5.1.4 Detailed example

This paragraph completes the *Person* example. It illustrates the **composition** operator and the naming mechanism. Two specifications are discussed : the first one supposes the definition of the *Person* color graph has already been made using **defcolorgraph** and **deftransition** definitions ; the second one, does not. In both cases, the definitions of the *Age* and *Sex* c-graphs are, of course, supposed to exist (see figure 14).

a) first specification

Here, we suppose the definition of the *Person* color graph already exists. In this case, the only thing to do is to specify the *Sex* and *Age* superclasses (in this order), relying on the **composition** operator for identifying the first p-subgraph of the *Person* p-graph with the first superclass c-graph (*Sex*), andthe second p-subgraph with the second superclass c-graph (*Age*). Nothing else needs to be done since all remaining informations are part of the already known **defcolorgraph** or **deftransitions** definitions (notably, the *boy* and *girl* blend, as well as the *long-lived* transition).

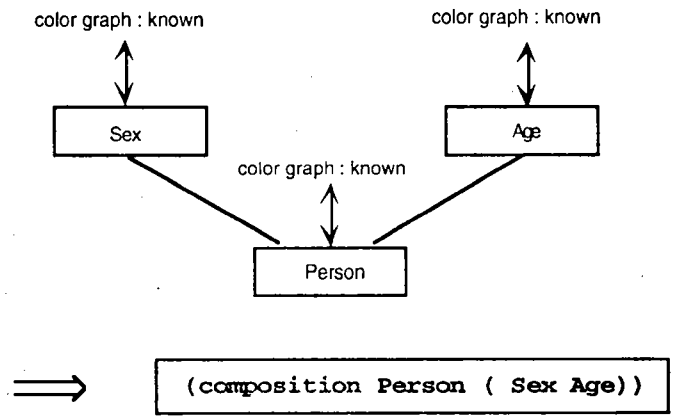


Figure 29.

(In the implementation part, this quite simple declaration will also imply the automatic inheritance of methods and memory representations attached to the color graphs of superclasses *Sex* and *Age*.)

b) second specification

Contrasting with the previous figure, next three ones do not suppose the existence of the *Person* **defcolorgraph**. Supplementary informations (vs. the *Age* and *Sex* c-graphs) need to be specified. Next figure expresses the situation and the declarations to be made.

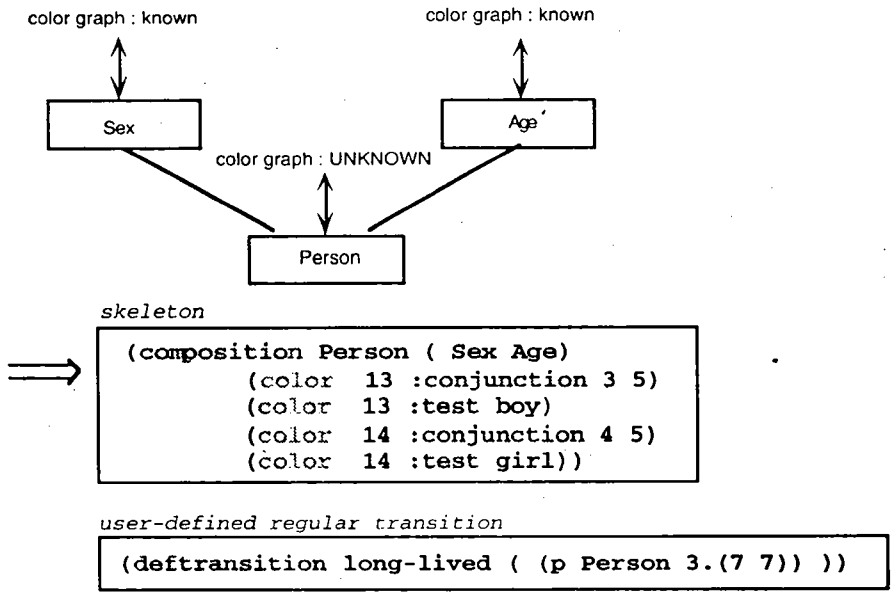


Figure 30.

Let's now explain these declarations step by step.

b.1) specification of the *Person* skeleton

Next figure shows the intermediate step. The analog of the **defcolorgraph**, a **setcolorgraph** operator is used : it lists the superclasses *Sex* and *Age* (in this order), then critical *Person* chromas. Color 1 is specified thanks to a **decomposition** construct, like in a **defcolograph**. Its pigments are in order : color 1 of the first superclass c-graph (*Sex*) and color 1 of the second superclass c-graph (*Age*). Blends *boy* and *girl* need also to be specified : except for the naming of pigments (here, replaced by the naming of colors in c-graphs of superclasses), this is done exactly like in the **defcolorgraph** of *Person*. [25]

skeleton

```
(setcolorgraph Person ( Sex Age)
        ( color   1 :decomposition 1 1)
        ( color  13 :conjunction 3 5)
        ( color  13 :test boy)
        ( color  14 :conjunction 4 5)
        ( color  14 :test girl))
```

Figure 31.

The implicit naming conventions are the following : ids refered to by the **decomposition** and **conjunction** constructs are ids in the c-graphs of superclasses, the order of ids in these constructs being the same than in the list of superclasses. For example, the first (resp. second) '1' in the **decomposition** construct is interpreted as color 1 in the *Sex* (resp. *Age*) p-graph. In addition, pigments in the each p-subgraph of the composed p-graph are named (numbered) in the same order as in the color graph of the associated superclass, each p-subgraph being taken in the order of the list of superclasses. In our example, given the definitions of *boy* and *girl* blends (last four lines), this determines the same order of pigments and colors than in figure 7.

The skeleton definition in figure 30 directly derives from the preceding definition : the **composition** operator encapsulates the **setcolograph** and the definition of the initial color (color 1) using the **decomposition** construct. Obviously enough, if a **defcolorgraph** already defines the *Person* skeleton, then the last four lines of this **composition** are useless ; once simplified, we get the simple specification of figure 29.

b.2) specification of the *long-lived* transition

One problem now remains : the specification of the *long-lived* transition, i.e. the transition which cannot be inherited from the*Person* superclasses. The simplest thing to do here is to specify it using the *Person* ids : this is what has been done in figure 30 (same definition than in figure 9).

If we were willing to use the ids of the *Age* and *Sex* c-graphs, it will be necessary to conceptually specify a couple (superclass name, color id) for each source and destination. Because superclasses are known, the notation proposed in the next figure (**Person* instead of *Person*) simply indicates that ids are to be searched in the color graphs of superclasses instead of the composed p-graph.

user-defined regular transition

```
(deftransition long-lived ( (p *Person 2.(2 2)) ))
```

Figure 32.

As shown in this subsection, two directions need to be developed for expressing the composition of classes : operators for conveniently composing dimensions ; naming facilities. The first point is focused on in the next subsection ; the second one will be the matter of another report.

[25] If the **defcolorgraph** of *Person* exists, the underlying system will warn the user of a double definition (a single one is preferable) and check the **setcolorgraph** definition vs. the **defcolorgraph** definition.

5.2 OPERATORS FOR COMPOSING OR ADDING DIMENSIONS

This subsection comprises two parts : the first one is dedicated to the **composition** operator (cartesian product of all points present on each dimension) ; the second one, to the **derivation** operator[26] (addition of dimensions on one specific node of a color graph). Specific inheritances rules are listed when appropriate.

5.2.1 Composing dimensions

The first operator we consider is the **composition** operator introduced in the preceding subsection. It directly brings into operation the very basic idea of class inheritance. It composes the dimensions of two or more classes C_1, C_2,... (of degree N_1, N_2,...) into a (N_1+N_2+...) dimensions space. In fact, a local increment possibly with its own dimensions may also be added. (Note the increment of the *Person* example did not introduce new dimensions.)

This operator takes advantage of the **decomposition** construct that exists at the LOCAL level and somehow promotes it at the HIERARCHY level. The order in which these classes C_i, termed the **superclasses** of the composed class, are listed is important if a **masking** effect is sought. **Renaming** of nodes and transitions is possible if necessary.

To describe this operator, two steps are proposed : the first one uniquely considers the effect of composing (the dimensions of) several superclasses ; the second one, uniquely the effect of composing (the dimensions of) an increment with (those of) one superclass. Each step is introduced via an example.

a) Composing the dimensions of several superclasses

For the sake of the demonstration, the example we consider is a bit more complex than the *Person* example already discussed. The new one is meant to illustrate masking and renaming facilities[27]. It deals with the composition of a particular type of objects behaving both like a *Stack* instance and like a *Queue* instance. One can *push* an element on its front or *enqueue* it at the rear ; however, elements are only obtained from the front using a *pop*. Let's name the class of such objects *STQ*. Our goal is to compose this class using the *Stack* and *Queue* classes as seeds.

a.1) a cartesian representation

The next figure represents the *STQ* p-graph as well as the *Stack* and *Queue* c-graphs.

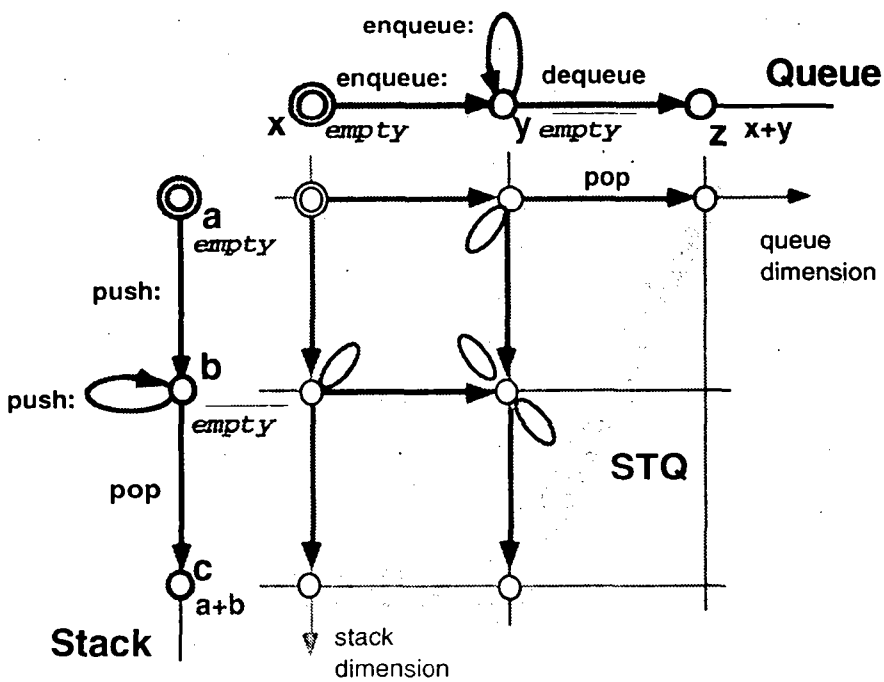


Figure 33.

---

[26] The design of the **derivation** operator is the subject of companion paper n° 2.

[27] These facilities could have been introduced directly at the p-graph level. However, the problem is better posed when combining superclasses : because these ones do pre-exist, tools are crucially needed to carve the intended result out of them.

The *Stack* (resp. *Queue*) colors and transitions exist along on the vertical (resp. horizontal) axis. (The *Stack* color graph was already shown. The *Queue* color graph is topologically similar.) An ephemere color (like *c* in *Stack* and *z* in *Queue*) normally corresponds to a cloud of several basic colors (like *a* and *b* in *Stack*, or *x* and *y* in *Queue*) : in the figure, it is represented as a point on the axis of the basic colors.

Composing these two c-graphs normally produces nine points. As a matter of fact, only those marked by a small bubble corresponds to reachable states in *STQ* : this comes from the specification of *STQ*. Consider the situation where a strictly positive number of elements have been pushed and a strictly positive number of elements have been enqueued, all in a *STQ* instance. Suppose now a *pop* is sent to this instance : this *pop* should return a pushed element, not an enqueued one. If *pop* messages continue, all pushed elements will be returned. After that, *pop* messages will return enqueued elements (supposing no *push:* occurs in between). In other words, a *pop* sent to a *STQ* instance is to be interpreted as a (*Stack*) *pop* unless the instance does not hold a pushed element any longer, in which case the *pop* is interpreted as a (*Queue*) *dequeue*. Masking and renaming facilities are thus required.

Next figure represents the same *STQ* composed behaviour using an extra dimension for ephemere nodes.

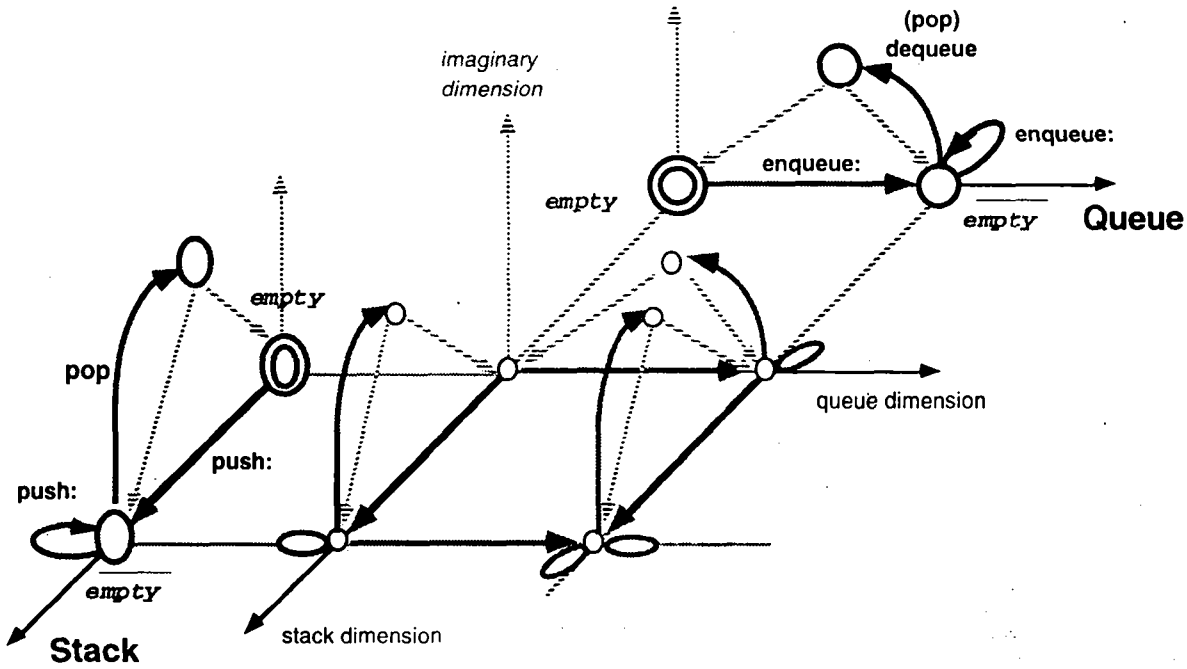


Figure 34.

Next two figures show the composition of the *STQ* color graph both in textual and visual forms. *Stack* preceedes *Queue* in the superclasses list ; this order is taken adavantage of for masking.

a.2) textual specification

It is organized as follows : the subclass is first named, then comes the superclasses list (with optional renaming and masking declarations), and finally the usual body of a color graph definition (here, it is void).

Consider the superclasses list :

– *Stack* is followed first by a list of ids (for naming each of its states unambiguously), then by two declarations stating that any *pop* or *top* transition of *Stack* masks any similar transition in a superclass placed afterwards (here, *Queue*);

– *Queue* is followed first by a list of ids for renaming its states, then by a declaration renaming its *STQ* transition in *pop*.

```
(composition STQ ( (STACK
                        (pop :masking)
                        (top :masking))
                  (QUEUE
                        (dequeue:now pop))))
```

Figure 35.

Any transition which is renamed or declared as masking in a given class should effectively exists in this class, be it locally defined or inherited.

a.3) visual specification

The figure below depicts the **composition** of the *STQ* color graph using the *Stack* and *Queue* color graphs. Renaming and masking are respectively encoded by mentioning the initial name between parentheses and by surrounding the transition name with a tiny rectangle. Note that the use of *empty* is not ambiguous : in the left (resp. right) subgraph, it corresponds to empty in a *Stack* (resp. *Queue*).

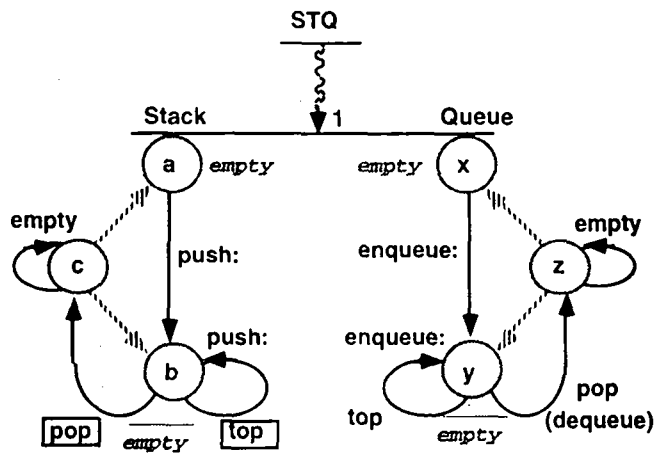


Figure 36.

The interested reader will find in appendix (figure A.1) the c-graph resulting from the expansion of this p-graph. Due to the masking effect of *pop* and *top* in *Stack*, this c-graph is naturally unsymmetrical.

a.4) additional class inheritance rules

As far as inheritance is concerned, the basic rules for class inheritance simply needs to be adapted to renaming and masking. Both are done dynamically. Let's suppose a message M is sent. Renaming is obtained by adopting the old name in place of the new name as soon as the superclass in question is entered : the old name is considered inside the whole hierarchy of this class (except if it is itself the result of an inner renaming inside this same hierarchy) ; it is abandoned on quitting it. Masking is obtained by abandoning the search once leaving the hierarchy of the first superclass where M is declared as masking (by construction, a M transition exists in this superclass, either inherited or locally defined).

Suppose, for example, *Stack* inherits from *Bag* and *Bag* from *Object*, with *Bag* providing *get* transitions and *Stack* remaning *get* into *pop*. Further suppose a *pop* transition is searched in *STQ* : when entering *Stack*, *get* will be searched in place of *pop* ; *get* will not be found in *Stack*, but in *Bag*. Since the *pop* transitions of *Stack* are declared as masking in *STQ*, *pop* will not be searched in *Queue*. Thus, the *pop* transitions of *STQ* result from the *get* transitions of *Bag*.

b) Composing the dimensions of a superclass with those of an increment

b.1) Principle

Till now, a new class was built by composing the dimensions of its superclasses, without adding any increment (preceding subsection) or with one that introduced no new dimensions, but only blends and a new transition (cf. the composed *Person* p-graph). We now consider the effect of a different form of increment, a local description with its own dimensions, and propose a syntax for handling it conveniently. Because the handling of multiple superclasses has already been examined, a single superclass is considered here.

In a first step, the problem under consideration is formalized as a composition of two superclasses (an instance of the problem previously discussed) : the increment is isolated into a new superclass (this corresponds to what can be —and would be— done by a programmer given the present primitives) ; then, it is composed with the regular superclass. As a result, the dimensions of both superclasses are composed. In a second step, the declaration of the increment as a superclass is encapsulated into the **composition** itself. This requires the observation of a few additional conventions.

b.2) Example : the *Bag* class

First, let's consider the *Object* c-graph which is inherited by the *Bag* p-graph. This c-graph exhibits one and but one dimension (say *object*) with a unique color (say α) the condition of which is *t* (systematically true). The *object* dimension is **abstract** (no memory representation). Next figure shows two definitions of α : the first one explicitly mentioning the condition and the :**abstract** keyword ; the second one specifies α as a **property**, which is equivalent. (Other examples of properties (*LIFO, FIFO*) will be given shortly.) Normally, the :**property** keyword is followed by a list of all the transitions that satisfy the property : here this list is absent, meaning that all the *Object* methods satisfy the property (default case). For our purpose, *Object* exhibits along its unique dimension (*object*) only one regular transition, the *print* one. Because the *Object* class is inherited by all classes, the *object* dimension is present in every object, and so the *print* transition is valid for any object. In our example, the *print* transition is inherited by the *Bag* class.

```
(defcolorgraph Object                    | (defcolorgraph Object
    (color α :test t :abstract))         |     (color α :property))
-----------------------------------------+-------------------------
(deftransition print ( (x Object  (α α)))
```

Figure 37.

Concerning the *Bag* color graph, we want it to exhibit four regular transitions : *print* (to be inherited from *Object*) ;*put:*, *get* and *empty* (to be locally defined). The *Bag* color graph has a topology similar to the *Stack* color graph.

Next figures show the two steps as defined above. First, to obtain the desired color graph, we combine in a two-dimensions space the single dimension of *Object* with the single dimension of a virtual superclass representing the increment. Next figure shows this in textual form. The figure afterwards displays the visual form and the combined color graph. This one is like the virtual superclass graph and has in addition the *print* transition attached to its ephemere color[28]. Note that the virtual superclass is first in the list of superclasses of the **composition**.

```
(defcolorgraph Bag-Virtual-Superclass
    (color a :test empty)
    (color b :test-not empty)
    (color c :selection a b))
(deftransition put: ( (x Bag-Virtual-Superclass  (a b) (b b)) e))
(deftransition get  ( (x Bag-Virtual-Superclass  (b c)) ))
```

```
(composition Bag ( Bag-Virtual-Superclass Object))
```

Figure 38.

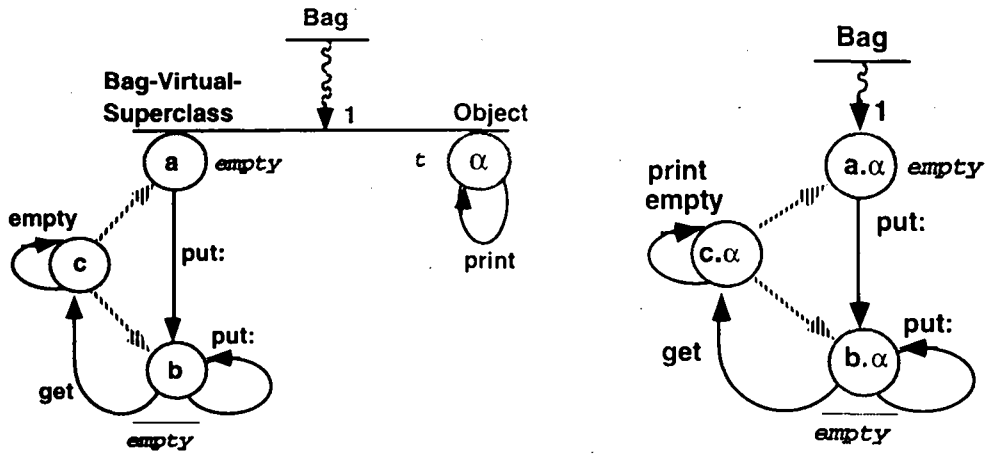


Figure 39.

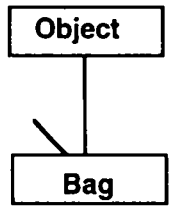
---

[28] The *Object* class, via the *object* dimension, factorizes for all classes a number of common transitions. In our example, *print* illustrates that. A way to understand *print* —and, by the way, all these transitions— consists in de-factorizing, i.e. in reintegrating the *object* dimension into class *Bag* (the *Object* class disappears in the transformation). The *print* transition is attached to pigment α and is given a non void clause : *[c]*. Alternatively, the *print* transition can be attached to pigment *c* and be given the non void clause *[α]* ; or, it can be attached (without any clause) to the blend resulting from the conjunction of α and *c*. (This understanding can be enlarged to the *print* method. See 7.2.2.b).

Now comes the second step. The virtual superclass definition is put inside the **composition** : the increment appears in the form of a collection of pigments (they denote a new dimension), whereas a collection of blends appeared in the composed *Person* p-graph (they added no new dimension). A more complex example would exhibit both pigments and blends. Next figure shows the source code of *Bag* . The figure afterwards gives a possible representation that symbolizes the two dimensions involved in *Bag* : the oblique line is reminiscent of the *Bag-Virtual-Superclass.*

```
(composition Bag ( Object )
    (pigment a :test empty)
    (pigment b :test-not empty)
    (pigment c :selection a b)))

(deftransition put: ( (x Bag (a b) (b b))  e))
(deftransition get  ( (x Bag (b c)) ))
```



Figures 40 & 41.

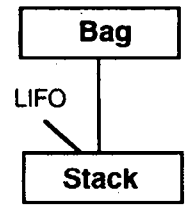
Additional conventions implicitely apply in the above *Bag* definition : they enable the two **deftransitions** to unambiguously specify their source-destination pairs using the local pigment ids. More generally, the naming can be done relatively to any ancestor c-graph ("**internal naming**")provided that this naming is unambiguous. A distinct report will describe this convention and, more generally, how node names may be specified simply and unambiguously.

b.3) Taking into account special properties

Typically, a local increment —like the *Bag* one— is composed of a few nodes with transitions between them. Yet, in some cases, the increment properties cannot be formalized as simply : in this case, they are identified using **property** nodes (see the *Object* class). This idea is adapted from the property identifiers introduced by P. America [America, 1990], p. 71. The *LIFO* organization of a *Stack* is such a kind of property. From an inheritance point of view, a property identifier corresponds to a new dimension with a single node (pigment) the condition of which is *t* (systematically *true*). This dimension is abstract (no memory representation). The node appears in transition clauses as a reminder of the required property : following the **:property** keyword, is a list of methods for which the property is automatically set to *true.*[29]

```
(composition Stack ( (Bag
                        (put:  :now push:)
                        (get   :now pop)))
                   ( pigment LIFO :property
                                    (push: pop top) ))

(deftransition top ( (s Stack (b b))))
```



Figures 42 & 43.

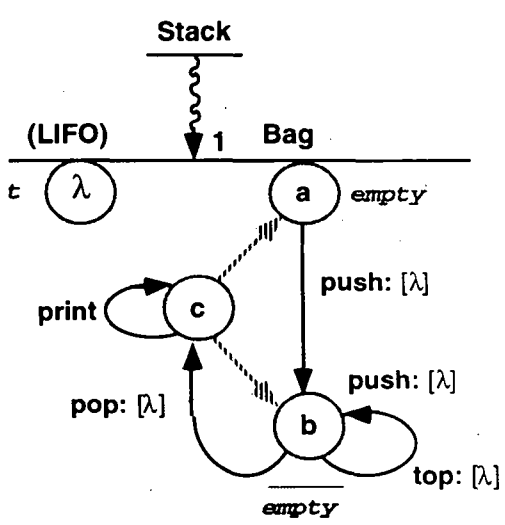
In our examples, *Stack* has a *bag* dimension (it inherits from *Bag*) and a *LIFO* dimension ; similarly, *Queue* has a *bag* dimension (it inherits from *Bag* too) and a *FIFO* dimension : this makes both the *Stack* and *Queue* color graphs topologically similar to the *Bag* c-graph, yet distinct from it and distinct from each other.

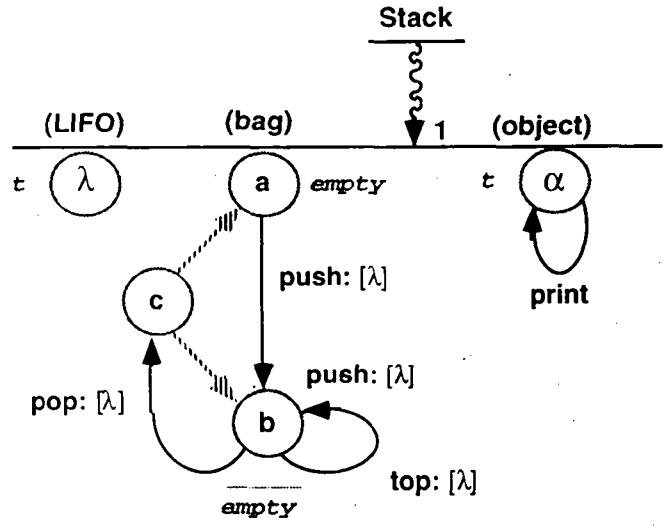
Figure 42 shows the *Stack* p-graph with the *LIFO* property declaration in textual form. In addition, transitions *put:* and *get* are renamed ; the *print* transition is implicitely inherited ; the *top* circular transition is added. Like the declarations of *put:* and *get* in *Bag* (see figure 40), the declaration of *top* illustrates the internal naming, yet at a deeper level (recursive aspect) : conceptually, the local description of *Bag* is considered as a superclass of *Bag*, i.e. as a superclass of the superclass of *Stack*. (The naming of the *b* pigment is obviously not ambiguous.) Figure 43 is the analog of figure 41 : it represents *Stack* with its superclass *Bag* ; the *LIFO* property of *Stack* is also shown as a virtual superclass.

Next figure shows the *Stack* p-graph in visual form. Note the *push:*, *pop* and *top* transitions are all constrained by pigment λ, the one representing the *LIFO* property : all these transitions were named in the property declaration of *LIFO* (see preceding figure). The figure afterwards shows the *Stack* p-graph once *Bag* has been expanded. Like superclasses, dimensions are named above the decomposition bar ; but, being different from superclasses, their names are shown between parentheses.

---

29 The solution chosen here for renaming is general (renaming may also be done simply in case of name clashes). Yet, one may consider that the renaming part could semantically be attributed to the *LIFO* property. This leads to devising a more appropriate syntax : the renaming being specified via the *LIFO* property, the subgraph inherited from *Bag* will appear directly as it is in *Bag*, i.e. without changing the names of the transitions *get* and *put:*.





Figures 44 & 45.

b.4) Additional class inheritance rule

Considering the modelling of the local increment as a virtual superclass, the important point here is to consider this virtual superclass before all other superclasses so as correctly take into account masking -if transitions defined in the increment are (explicitely or implicitely) specified as such.

5.2.2 Adding dimensions in one point

The above defined **composition** operator basically combines all the dimensions of the superclasses (plus those of the local description if it exists) : if N_i points exist on each i dimension, we obtain ΠN_i points as a result. Some of these points may be eliminated using the **:masking** facility as illustrated in the *STQ* example. However, it may occur that we are not interested in systematically combining the dimensions of one superclass with those of the other superclasses. Let's illustrate this with an example.

a) The *Bounded-Stack* example

a.1) Problem description

A *Bounded-Stack* instance behaves almost like a *Stack*. It should thus be possible to get a description of the former by changing a bit the behaviour of the latter. As shown above, the color graph of a *Stack* exhibits three colors[30]. Its behaviour is altered only in one color : when a *push:* message occurs in color 2 (*not empty*), a test should occur that possibly moves the instance state to *full*. In *full*, the *push:* transition is marked as **unwilling**. (This means the usual effect of a *push:* message is cancelled (redefined) ; however, the transition still exists to respect usual constraints on transitions. For more details, see companion paper 2.) The *Bounded-Stack* instance behaviour is unchanged in color 1 and 3. Using the **composition** operator would thus be fairly inappropriate. Next figure explicits the expected behaviour in a 2-dimensions space when the bound (maximum number of elements in the *Stack* instance) is strictly greater than 1.

[30] For the purpose of simplicity, we use here the first *Stack* color graph . The second one (similar to the *Bag* one, with the *print* transition attached to *Object*) can also be used : the figure is then to be (formally) drawn in a 3-dimensions space.

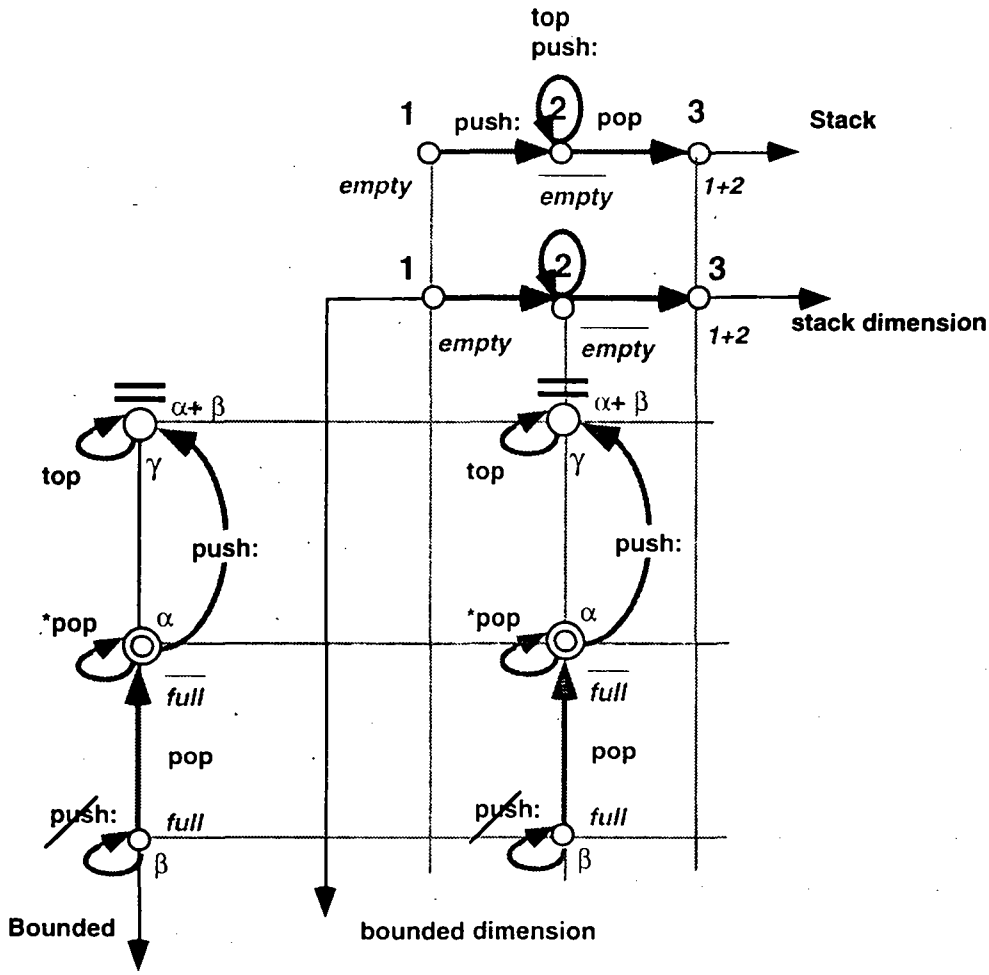


Figure 46.

Next figure shows the same assembly of the *Stack* and *Bounded* color graphs in a 3-dimensions space : an extra dimension is used for ephemere nodes.

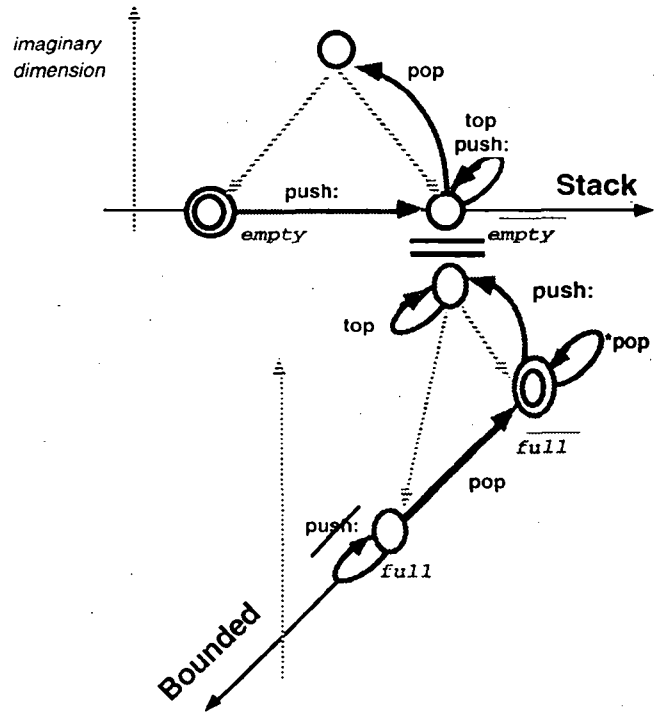


Figure 47.

a.2) The derivation construct

To model such a situation, COP provides the **derivation** construct (see reference [Borron, 1995b] and, for a detailed study, companion paper n° 2). Using figure 46, this construct can be interpreted by considering the *Stack* mini-token. This one moves along the horizontal axis. When present in color 2, another mini-token is created : this one is the *Bounded* mini-token ; it moves along the vertical axis erected in color 2. This mini-token is created in γ. γ is the source of a selection on α (*not full*) and β (*full*). The mini-token automatically moves to *not full* because *not full* is declared as being the initial node of the selection. When the *Bounded* mini-token is in *not full*, a **starred** *pop* message makes the test *not empty* to be checked. If the condition is not verified, the *Bounded* mini-token disappears ; given the result of the test, the *Stack* mini-token moves to *empty*. If the condition is verified, the *Bounded* mini-token stays in *not full*. A *push:* message makes it move to the selection root. From there it goes automatically either to *full* or to *not full*. In *full*, besides the special behaviour in case of a *push:* message, a *pop* message will make the *Bounded* mini-token to move back to *not full*..

The derivation construct manages these constraints while being simpler than a constrained p-graph and more modular than a a c-graph. Companion paper n° 2 discusses in length the *Bounded-Stack* example.

b) Mixins classes

b.1) First specification

The *Bounded* supplement of behaviour is useful not only for describing the behaviour of a *Bounded-Stack* instance, but also that of a *Bounded-Queue* instance (and many others). Hence, it appears a good idea to extract it from the *Bounded-Stack* color graph and give it a name to reuse it conveniently whenever appropriate. We thus capture this *Bounded* behaviour in what we call a **mixin** (since the term has already got such a meaning). Still because the purpose of a mixin is to be reused, transition and/or condition names are given abstract names (i.e. *push:* is replaced by *put:* ; and *pop*, by *get*). The color graph depicted in the next figure corresponds exactly to the above cartesian representation. A mixin color graph cannot be instantiated by itself : no creation transition exists ; in addition, it is not included in the list of color graphs having predefined instances (such like *Integer*). The double bar indicates that its destiny is to be used in a derivation. The stroke above the circular *put:* transition means in practise that no *put:* message is acceptable in the *full* state (unwilling transition). The MAC table, which stores the attachment constraints, belongs to the mixin specification.

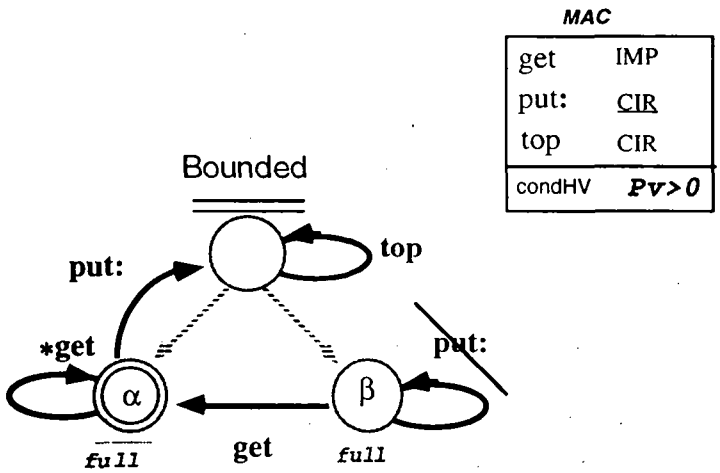


Figure 48.

b.2) Parameterized mixin

The *Bounded* behaviour described so far is valid only when K>1, where K is the bound (i.e. the maximum number of elements that can be stored in a given *Bounded-Stack* instance). To make the mixin as general as possible, an additional mixin is provided for K=1. The parameter K is added to the MAC and each provided mixins is attached its own condition depending on the K parameter. The underlying system builds automatically a parameterized mixin on top of that and knows how to use it efficiently (the adequate mixin is chosen at creation time). Next figure shows the corresponding specification of the *Bounded* mixin.

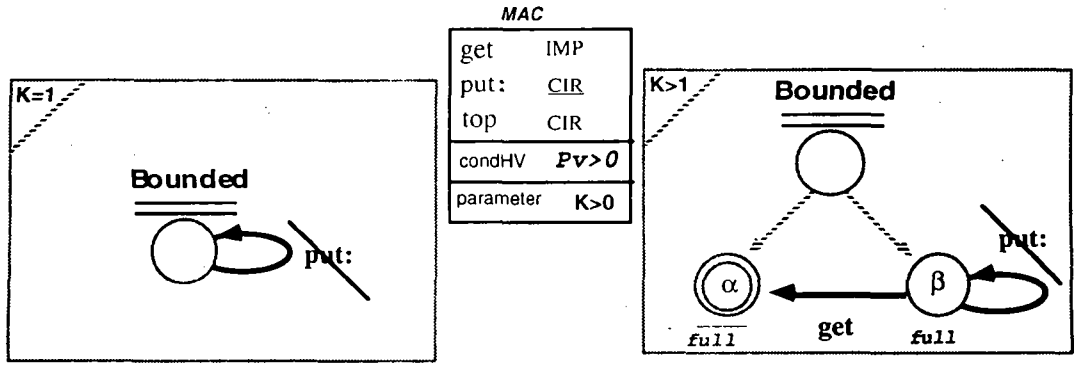


Figure 49.

The first mixin simply makes the *put:* transition be unwilling. The second mixin is strictly equivalent to the one shown in figure 48 ; yet, it is reduced to an extreme simplicity due to default rules established in companion paper n° 2.

b.3) Textual specification

Next figure shows the textual syntax for N>1 (attachment constraints are not depicted). The mixin skeleton is a (possibly multi-level) selection : the root node (numbered 1 by default) as well as the upper level selection are implicitely declared. This constrains (and simplifies) the specification : the user is only asked to describe the other nodes.

```
(defmixingraph Bounded
     (  (pigment α :test-not full)
        (pigment β :test full) ))

(deftransition put: ( (s Bounded (β β)) e) :unwilling)
(deftransition get  ( (s Bounded (β α))) )
```

Figure 50.

b.4) Derived color graph

Next figure shows the*Bounded-Stack* color graph in visual form (K>1). Note that it can be made extremely concise (middle part of the figure): between the double-bar separating the base and mixin color graphs, *not empty* (or 2) corresponds to the **hook** node. The right part of the figure is taken from companion paper n°2. It shows the equivalent c-graph obtained by expanding the left part according to our rules for mixin elaboration : the interested reader may easily check no useless testings need to be done at run-time (in spite the graph is obtained from a parameterized mixin);

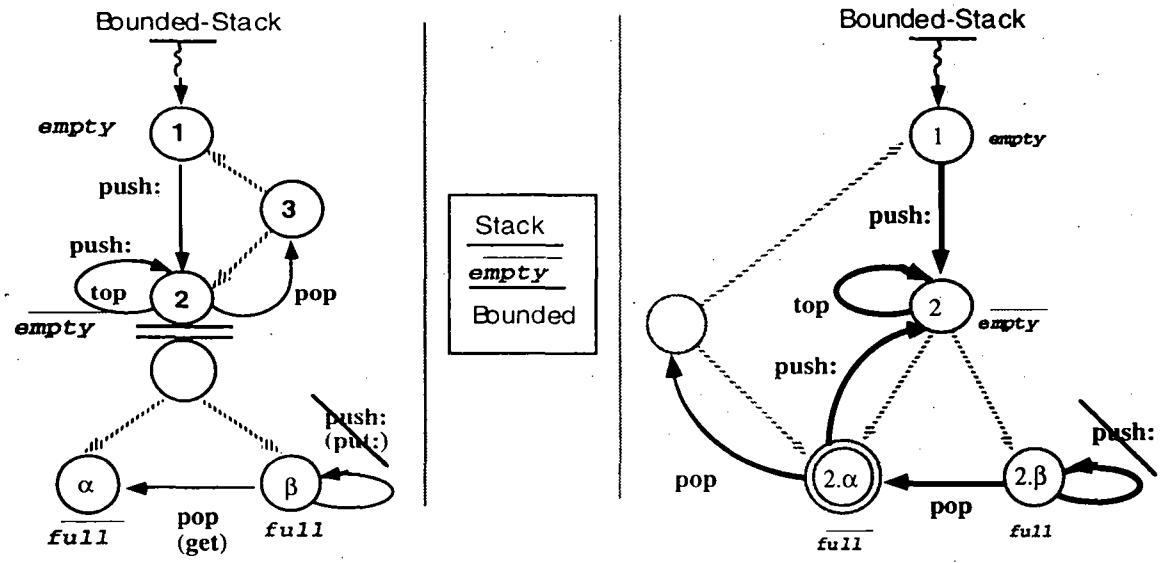


Figure 51.

Next figure shows the *Bounded-Stack* color graph in textual form. The **derivation**, like the **composition** operator, has a list of superclasses. Note that the mixin class is named before the base one. (This is preferable, but not mandatory.)

```
(derivation Bounded-Stack ( (Bounded (put: :now push:)
                                     (get  :now pop))
                            (Stack 2)))
```

Figure 52.

b.5) Additional class inheritance rules

Obviously enough, the algorithm which extends local inheritance to class inheritance works accordingly well in this *Bounded-Stack* example. We simply want the definitions made in the mixin graph (*Bounded*) may refine similar definitions at the hook node of the base graph (*Stack*) : (1) the mixin graph is visited before the base one ; (2) the transitions defined in the mixin graph are implicitely declared as masking ; (3) the validity of the mixin graph is limited (the base mark should be in the hook node). First two rules makes inheritance in this case not especially particular. Last rule is implemented by making the mixin graph a constrained subtree (see companion paper n° 2). The declaration of the mixin class before the base class in the list of superclasses is easy to check and to correct : the underlying system knows which is a mixin and which is a base).

As an example, let's consider an instance of *Bounded-Stack* again. In colors 1 and 3, *Bounded* is not active, thus the instance behaves exactly like a pure *Stack* instance. In any state (*, 2) (where * is any pigment of the *Bounded* mixin), *Bounded* masks *Stack* : we thus obtain the desired behaviour.

Next figure abstracts the relationship between the three classes as a small hierarchy, like in OOP : note this manner is quite abrupt since it omits to mention the hook node, an important detail in fact (compare with figure 51).

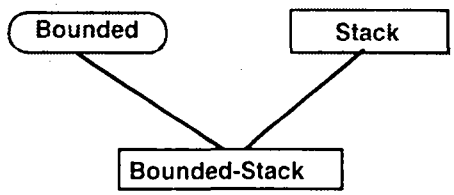


Figure 53.

PART B : INHERITANCE OF IMPLEMENTATIONS

This part explains first how an implementation can be attached to one color graph (local implementation), then how an implementation can be distributed into a hierarchy of color graphs (hierarchical implementation). Hence, respectively the local inheritance rules for implementations and the class inheritance rules for implementations. This work is done considering first a Smalltalk-style implementation (no combination, systematic masking) and then a CLOS-style implementation (systematic combination except in case of explicit masking). The solution finally proposed encompasses in fact both styles.

6. IMPLEMENTATION OF A COLOR GRAPH

This section relates to the second question of the introduction (cf. subsection 1.1). First of all, it states how an implementation is attached to a color graph. Then, it shows how the LOCAL INHERITANCE rules for TRANSITIONS can be exended to IMPLEMENTATIONS.

6.1 CONCRETE IMPLEMENTATION

A color graph represents the complete interface of a class, including its ancestor classes. It may well be implemented in but one class. We term **concrete** implementation the attachment of methods and memory representations to a color graph. The result is termed an **augmented** color graph.

Two novelties distinguish COP from OOP at this level : the possible attachment of several memory representations to a class ; the attachement of two methods (termed "micro-methods") to a transition.

6.1.1 Memory representation

a) Principle

A memory **representation** may be attached to each color or pigment if not abstract (i.e. if not belonging to an abstract dimension). A memory representation is made of a number of **cells** (slots in CLOS wording ; instance variables in Smalltalk wording). Changes of representations (ex. : between two colors, two pigments or two palettes) and combinations of representations (ex. : to get the representation of a blend made of two pigments) are done automatically. Note a change of representations (**change-rep**) between two chromas is similar to a **change-class** in CLOS.

b) Example

As an example of different memory representations in a same color graph, one may well consider the following choice for a *Stack* instance :
(a) no cells at all in the *empty* color ;
(b) an *elements* list in the other two colors.

```
(defclass STACK ()
    ( (colors 1 NIL )
      (colors 2 3 (elements :accessor elements
                            :initform ())))
```

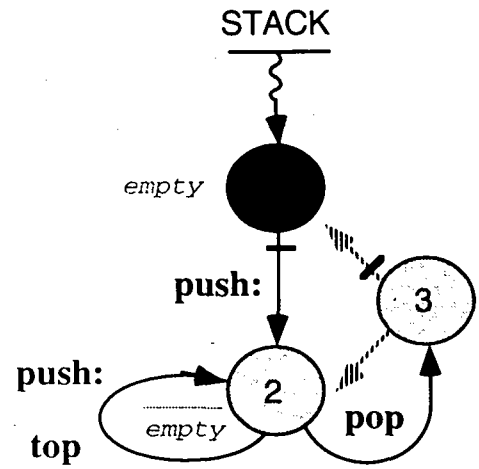


Figure 54.

In the previous figure, a specific pattern is used for each different memory representation ; a small bar represents an automatic **change-rep**.

6.1.2 Micro-methods

a) Principle

Two methods may in principle be attached to one transition : a **pre-method** at the transition source ; a **post-method** at the transition destination[31].

Next figure illustrates this using two small circles : a white one at the source ; a black one at the destination.

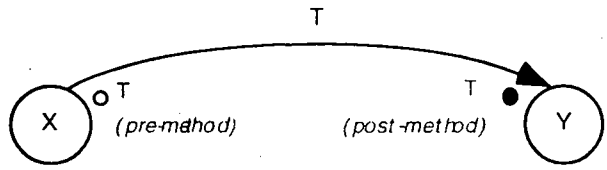


Figure 55.

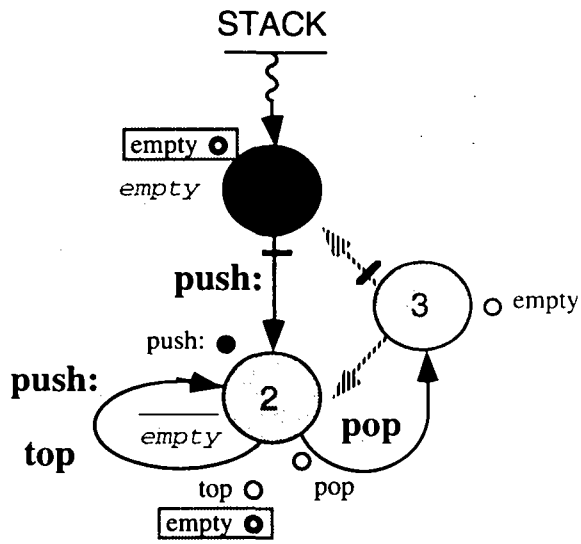
We use the term **micro-method** (or, more loosely, method) when we do not want to make a distinction between the pre- and post-method ; or when this is not necessary, the context being unambiguous.

b) Examples

Frequently enough, a pre-method alone convenes : this matches the OOP case. Sometimes, a post-method alone is a better choice ; other times, a couple of pre- and post-methods offers a better modelling[32]. Let's illustrate the last two cases.

b1) a single post-method

The second case is examplified with the *Stack* class again. Using the *Stack* representations described above, a pre-method appears absolutely useless to implement the *push:* transition in the *empty* color (a cell is needed to hold the pushed element, but this cell does not exist in the *empty* color) : a lonesome *push:* post-method is thus used.



user defined:

```
; color 2
(defpostmethod push: ( (s STACK 2) e)
        (push e (elements s)))

(defmethod pop ( (s STACK 2))
        (pop (elements s)))

(defmethod top ( (s STACK 2))
        (car (elements s)))

; color 3
(defmethod empty ( (s STACK 3))
        (null (elements s)))
```

Constants methods for empty *in colors 1 and 2 (inside grey bordered rectangles) are automatically generated.*

Figure 56.

---

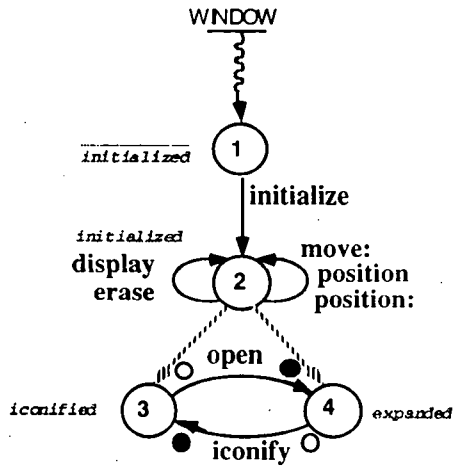
[31] Method qualifiers (cf. for example, the *:before, :after, :around* and primary methods of the standard method combination of CLOS) constitute a different, orthogonal issue.

[32] The distinction between the pre- and post-methods gets blurred when the same representation is used by the source and destination nodes of the transition, i.e. when no representation conversion exists (this is notably the case of a circular transition). Yet, in a number of circumstances, it may be conceptually clearer to still distinguish a pre-method and a post-method.

In the preceding figure, a unique post-method happens to be used by both *push:* transitions : no *push:* pre-method exists. The reason is simple : (1) this is the natural implementation of the first *push:* ; (2) this *push:* post-method alone also convenes for the circular *push:* transition (attached to the *not empty* color), the unique constraint being to implement this transition using a pre-method and/or a post-method.

b.2) a pre-method + a post-method

The third case is demonstrated with a *Window* class having a color *iconified* and a color *opened*[33]. The *open* transition from this first color to the second one is obtained with a pre-method for erasing the icon and a post-method for displaying the window itself (the *iconify* transition does the opposite using also a couple of pre- and post-methods). Such a neat modelling cannot be rivalled by traditional OOP.




```
(defmethod open ( (self WINDOW 3))
      (premethod (erase self))
      (postmethod (display self)))
```

Figure 57.

For more details about the syntax of micro-methods (notably, concerning arguments), refer to [Borron, 1995b].

6.2 LOCAL INHERITANCE RULES FOR CONCRETE IMPLEMENTATIONS

This subsection extends the LOCAL inheritance rules : given the ones for TRANSITIONS, it devises new ones for IMPLEMENTATIONS.

More precisely, given class *C*, its augmented color graph, and a *C* instance in a certain state, we are interested in finding what memory representation is to be considered for this state, and what pre- and post-methods are to be activated upon the reception of a valid message *m*. We term **item**, the memory representation, pre-method or post-method we are looking for. These items are to be searched in the ancestor-graph of the instance in question. Under these circumstances, we say that a **dimension is to be inspected** when the item is to be searched in the restriction of the considered ancestor-graph to this dimension.

In the following, we first intuitively examplifies the relationship between LOCAL inheritance rules for TRANSITIONS and LOCAL inheritance rules for IMPLEMENTATIONS. Then, we specify the latter rules.

6.2.1 Introduction

a) Motivation

We already know the LOCAL inheritance rules for (regular) TRANSITIONS :
— regular transitions are **inherited** along the reflex transitions (except along those flowing from a decomposition) ;
— in case of a composite transition, the destination is obtained by **composing** the elementary destinations ;
— an inherited i-circular transition is circular ; otherwise, the destination is to be computed dynamically ;

Knowing that TRANSITIONS are inherited, it is very temptating to devise rules for inheriting IMPLEMENTATION **items** (memory reprsentations, pre- or post-methods) along the same reflex transitions.

---

[33] This example is also drawn from [Chambers, 1993], subsection 3.5, page 219.

b) Example

Consider, for example, a memory representation defined in color 3 of the *Stack* c- graph : it appears natural to inherit this one in colors 1 and 2. Masking may also be demonstrated in this example : it occurs if a decision is made to choose a specific representation for color 1 (notably, one that offers no cells in this color). (See next figure.)

```
(defclass STACK ()
    ( (colors 1 NIL )
      (colors 3 (elements :accessor elements
                          :initform ()))))
```

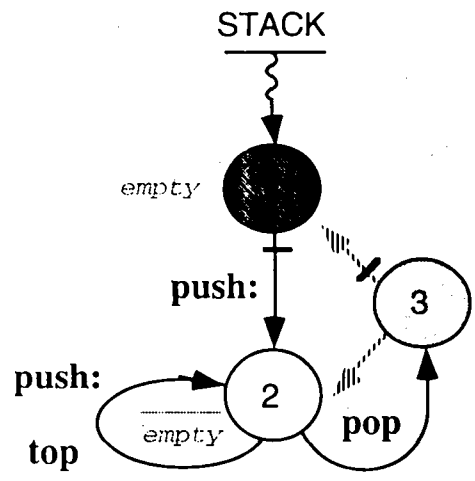


Figure 58.

c) Tackling strategy

Somehow, we are facing –at a different level– a problem encountered in OO languages with class inheritance. The ancestor tree plays here the role of the class hierarchy. Being interested in automatic combination of items, we first consider a **Smalltalk-style** implementation delaying a **CLOS-style** one to the end of PART C. In other words, we first focus on a style of implementation that, given some order, a priori systematically selects the first item satisfying some criteria and systematically ignores the next ones (implicit masking). Once this style of implementation will be fully described (i.e. when described in a hierarchy of classes), we will turn to the case of the a priori systematic selection and combination of items that satisfy the criteria.[34]

d) Proposal

The solution we propose is directly modelled from the handling of transitions (see subsections 4.1.2 and 4.2.2) :
— as for transitions, it consists in inspecting each dimension in turn and selecting each time one and only one valid item if it exists. (In fact, all dimensions need not be inspected systematically : for example, when searching a method *m*, are only **involved** the dimensions for which exists the transition *m* in question.) The rule is that each involved dimension should be satisfied once and only once ;
— as for transitions, items are searched along the reflex transitions when not found locally (local inheritance) ;
— as for transition destinations, a combination is to be donewhen the selection by dimensions has yielded several items. In a c-graph, no combination is ever necessary (one dimension). In a p-graph (N dimensions), a combination is often required. (Note that the combination of dimensions has nothing to do with the CLOS-style combination since the concept of dimensions is proper to COP.)

Next subsections progressively describe the solution, first in a c-graph (one involved dimension at most), then in a p-graph (N involved dimensions at most). Finally, possible pre-treatments are described. (The rules we propose are termed "search" rules instead of "inheritance" rules to avoid as much as possible a confusion with class inheritance.)

6.2.2 C-graph proposal

Here, the considered class *C* is defined along a single dimension (the state space has one axis). If not abstract, this dimension is systematically involved when looking for a memory representation ; for a pre- or post-method, it is involved only when the corresponding transition exists (if it does not : the implementation is erroneous vs. the specification).

---

[34] Justification : the Smalltalk-style is easier to tackle first than the CLOS one. An augmented color graph may be complex in term of items since it a priori concentrates in one place the combination of a number of incremental color graphs (one per class in the considered hierarchy). An implicit systematic masking thus brings a useful simplification when facing the implemetation problem for the first time.

a) The c-ancestor-tree local search rule

Here is the rule, also termed the "c-graph local search rule". In a c-graph, ***a single item is to be selected*** going up in the **c-ancestor-tree** of the considered chroma. Along each possible path, an item is chosen if one exists : this item is the first one encountered (the most specialized one) if several exist (masking effect). When the same item is found on different paths, it is retained but once. If this process results in the selection of one and only one item, this item is returned. If no item is selected, *nil* is returned (this may leads to signal error n°1). Otherwise (more than one item), an error is signalled (error n° 2).

nil result :
— when searching a representation, error n° 1 is systematically signalled ;
— when searching a micro-method, error n° 1 is signalled only when both the pre-method and the post-method searches have produced a *nil* result. (Note that two a priori different ancestor graphs are to be considered : one rooted at the transition origin for the pre-method ; one rooted at the transition destination for the post-method.)

b) Erroneous cases

b.1) two kinds of errors

— Error n° 1 : no valid item (memory representation, pair of pre- and post method) has been found in the c-ancestor-tree. This may be because no such item effectively exists in the c-graph or because the intended one was wrongly placed.

— Error n°2 : several paths converge to the considered chroma and two or more different valid items have been found. One item at least is wrongly placed or should be suppressed.

b.2) examples

Next figure shows the *Age* color graph with one *age:* method (correctly implemented), one *have-birthday* method and two *bedtime* test methods. The last two methods are incorrectly implemented. Both kinds of errors are examplified :
— *have-birthday* : the placement of the method convenes for *child* and *teenager* ; but does not provide a method for *adult* (error n° 1). The method should be moved to node 2 (*age*).
— *bedtime* : suppose the idea of the user is to specify a method for any possible substate (*child, teenager* or *adult*) of an initialized *Age* instance : this default method will be installed in color 2 (*age*). This choice is perfectly correct. Suppose now the user decides to refine the model and specifies a different *bedtime* method in node 3, the idea being to specify the same method for *child* and *teenager*. Then, this augmented c-graph is incorrect : in *child* and *teenager*, two different *bedtime* methods are in fact inherited and none masks the other one (error n° 2). The *bedtime* method attached to node 2 should be moved to node 7 (*adult*). Less elegantly, the same effect is obtained by moving the *bedtime* from node 3 to nodes 5 (*child*) and 6 (*teenager*) : in this case, when a search is done in node 5 or 6, the method found locally masks the method attached to node 2 (being attached to a substate of node 2, the former is a priori more specialized than the latter).

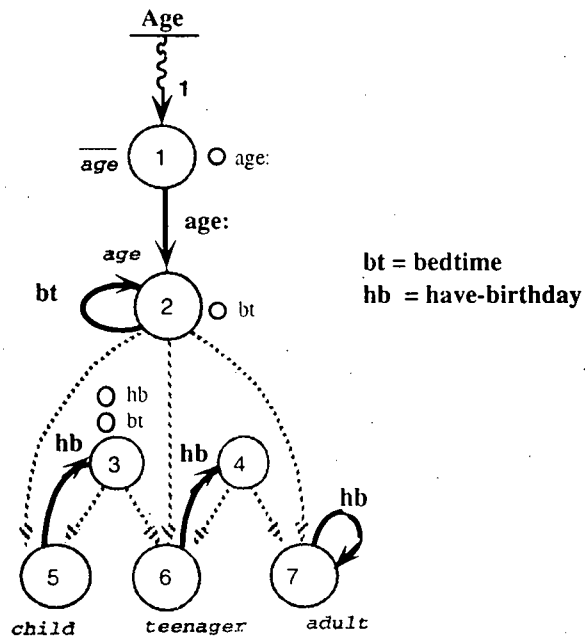


Figure 59.

c) Abstracting the correct results

Next figure abstracts the correct results for an instance of a class *C* in a certain state. The unique dimension *d* is represented by a single line since one and only one item (of a given type : representation, pre- or post-method) is selected even if several paths converge to the considered chroma. An item is depicted by a small bubble : a grey squared one for a memory representation (mandatory if *d* is not abstract) ; a labelled white rounded one for a pre-method, a labelled black rounded one for a post-method (at least one of the latters is required : hence three cases). The class *C* is represented by a large circle encompassing its items. These figures are termed **dimension diagrams**.

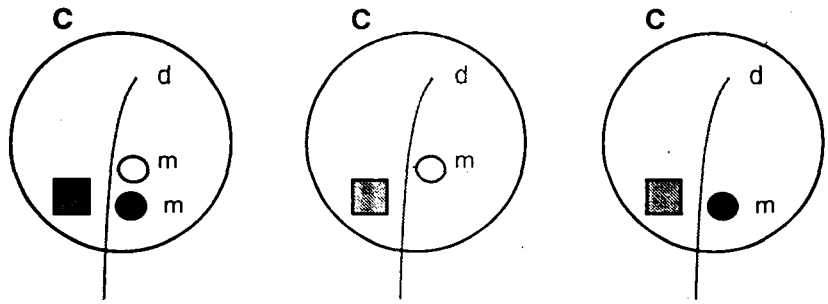


Figure 60.

When class inheritance will be considered, a line in a dimension diagram will generally traverse several classes, each one possibly adding its own items (those of a subclass masking those of a superclass in our Smalltalk-style hypothesis) : a line represents thus a **refinement path** vs. a given dimension. (Loosely speaking, we say —as above— that a line represents a dimension). In a state diagram like the one shown in figure 33, the line associated to a given dimension can be interpreted graphically as orthogonally traversing the various color graph contributions along the corresponding axis (from the least specialized to the most specialized) : in the cited figure, the *queue* line traverses the *Queue* c-graph and the *STQ* contribution (i.e. p-graph projection onto the *queue* dimension) for the *queue* dimension (similar remark for the *stack* line).

6.2.3 P-graph proposal

a) The p-graph local search and combination rule

a.1) basic idea

A p-graph exhibits N dimensions (in other words, the state space has N axes) whereas a c-graph holds but one. The idea is thus to consider each dimension in turn if it plays a role vs. the item to be searched (involved dimension) and to apply it a generalization of the "c-ancestor-tree local search rule" used in a c-graph. This generalization is termed the "p-ancestor-tree local search rule". Each time it is applied to a p-ancestor-tree, it normally produces an item. When all involved dimensions have been considered, the obtained items are then combined.

a.2) involved dimensions/p-ancestor-trees

The involved dimensions/p-ancestor-trees (be I their number) are those that play a role vs. the item to be searched.
-> all dimensions/p-ancestor-trees are involved when searching a representation (I = N)
-> when searching a micro-method, are only involved the dimensions/p-ancestor-trees from which the corresponding transition is (actually or virtually) inherited : only one in case of a simple unconstrained transition attached to a pigment (I=1) ; several ones otherwise (composite transition, constrained simple transition, or simple unconstrained transition attached to a blend (or a super-blend : see subsection 6.2.4 on pre-treatments)).

a.3) the p-ancestor-tree local search rule

In a p-graph, the "p-ancestor-tree local search rule" generalizes the "c-ancestor-tree local search rule" used in a c-graph. Basically, each involved p-ancestor-tree (restriction of the ancestor tree to one dimension) is considered in turn as a c-ancestor-tree (one dimension). The difference is that nodes in a p-ancestor-tree may be attached items that are valid not only for the considered dimension, but also for a number of other dimensions.

Each involved p-ancestor-tree should a priori provide one item and only one. This item should be the most specialized one (same rule as in a c-ancestor-tree). The degree of specialization of an item is due both to its attachment chroma and its clause. Here are two common subcases :

— if several items attached to a given node satisfy the search, the most constrained (and thus, specialized) one is selected. (If this item had been attached to a blend as well as the other candidate items found at the same place, this item would have been selected since its blend would have been visited before the other considered nodes : in other words, the considered item masks the other candidate items whatever the way it is attached).

— If two valid items having the same clause are attached to two nodes along a same path, then the lowest item is to be selected (this generalizes the c-graph case).

Note that an **involved p-ancestor-tree may well not provide a given item by itself** but may be declared to do so (possibly a posteriori) by an inspection in another p-ancestor-tree. Such a situation occurs for a constrained item : this one is physically attached to a node in one (or several) p-ancestor-tree(s) and virtually attached —by its clause— to one or several other nodes of other p-ancestor-trees.

Given all this, the algorithm should return the most specialized item in the p-ancestor-tree. If not unique, an error is signalled (error n° 2). If no item is found, *nil* is returned (this may leads to signal error n°1 : concerning methods, an error is signalled —as above— only when both the pre-method and the post-method searches have failed.)

a.4) duplicate elimination

At this point, each involved dimension has been examined (application of the "p-ancestor-tree local search rule") and —supposing that no errors have been detected— each one has provided a single item, the most specialized one. All these I selected items are listed in the order of the dimensions that provided them. Each p-ancestor-tree inspection being independent, a same item may have been selected as a result of n inspections ($1 \leq n \leq I$). Duplicated items ($n \geq 2$), if existing, are eliminated : only one occurrence, the first one in the order of dimensions, is retained per duplicated item.

a.5) combination of items

Each involved dimension is now satisfied once and only once. Retained items are then combined when more than one exist. The default combination method and the default combination order can be modified thru meta-object protocols possibly encapsulated under user-friendly options or primitives[35] (as CLOS does [Kiczales et alii, 1992]). The result of the combination is returned.

The default order to combine items is the order of dimensions in the decomposition construct. The default method for cells is the **concatenation** ; it is a **AND** for the micro-methods implementing a testing transition and a **PROGN** for the other micro-methods[36]. In case pre- and post-methods exist, pre-methods are executed first, then the post-methods (in reverse order) : in between, the instance state changes.

b) Abstracting correct results

Using dimension diagrams, next figure abstracts a few correct results for an instance of a class *C* in a certain state. Each diagram is drawn in case of three dimensions. Memory representations are not shown. In the first case, a specific pre-method exists for each dimension ; in the second case, a single couple of pre- and post-methods exists for the first two dimensions, and a pre-method for the third dimension ; in the third case, a pre-method exists for all three dimensions.

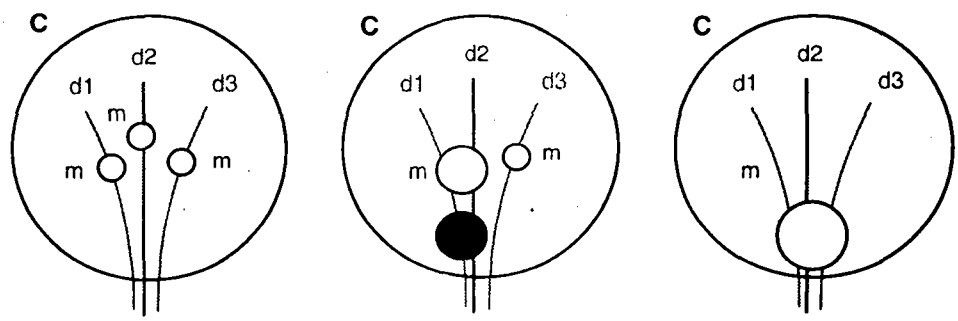


Figure 61.

35 The default micro-method-combination can notably be changed using a dedicated option in the **deftransition**. For example : (**deftransition** m (...) (**:micro-method-combination** +)). (This is analogous to the **:method-combination** option in a CLOS **defgeneric** definition.)

36 In less technical terms, they are executed in sequence ; the result of the last one is returned (it is the result of the combination).

c) Erroneous cases

As above, we can distinguish two kinds of errors, the same as above, yet generalized since we now consider several dimensions instead of a single one.

— Error n° 1 : no valid item (memory representation, pair of pre- and post method) has been found for (at least) one dimension of the p-ancestor-tree.

— Error n°2 : two or more valid items can be selected for one dimension of the p-ancestor-tree. These items satisfy the same number of dimensions but they differ by one or more dimensions.

Next figure illustrates both errors using dimension diagrams. In the first case, dimension *d2* is not satisfied ; in the second one, dimension *d2* is a priori satisfied twice with a first pre-method satisfying dimensions *d1* and *d2* , and a second one satisfying dimensions *d1* and *d3* ; in the third case, the pre- and post-methods are separately acceptable, but they differ on the aggregation of dimensions (pre- and post-methods may not exist at the same time, but when they exist their dimensions should agree).

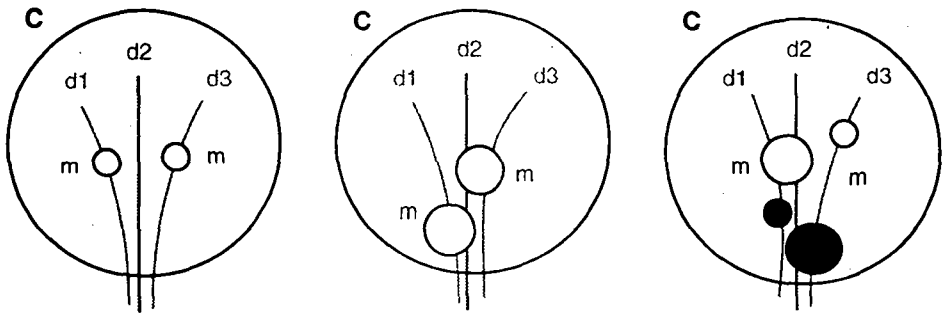


Figure 62.

d) More about duplicate elimination

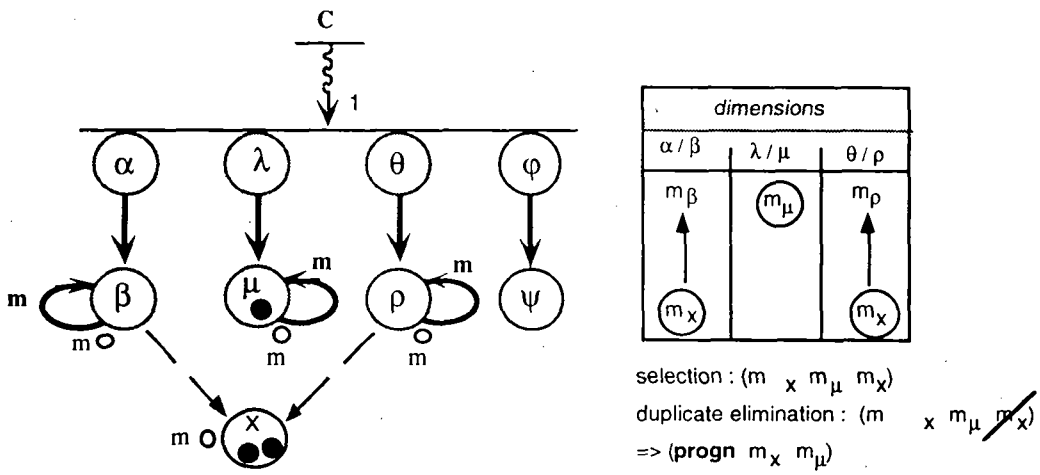


Figure 63.

The above figure illustrates duplicate elimination. Here, N is equal to 4 ; and I, to 3. In the considered instance state, when a *m* message occurs, "the p-graph local search and combination rule" is run. Dimension α/β is considered first, then dimension λ/μ and finally dimension θ/ρ. Selected methods (pre-methods) are, in order, m_x, m_μ and once again m_x. (m_x denotes the *m* method found in node *x*). (The same result would have been observed if the m_x item had been attached to either β or ρ –in place of *x*– and thus respectively constrained by either ρ or β : the m_β and m_ρ methods would have been masked).

Item m_x is thus duplicated. The above algorithm keeps but its first occurrence : the selected items after duplicate elimination are thus m_x and m_μ (in this order). How is this justified ?

(1) The duplicated item does require (informations stored for) both dimensions at once : unless uncorrectly modelled, this item cannot be split into two independent parts (ex. : considering a *Circle* instance, a *draw* method requires both the *radius* and *center* informations). This justifies that only one occurrence is to be kept.

(2) A choice is thus to be made among all its occurrences. In practice, the choice is restricted to the first or last one. Selecting the first one is coherent with the order of dimensions, an important point for enabling masking and refinement.[37] In the same vein, it will also be coherent with the CLOS-style implementation (see section 8 afterwards). (Note by the way that methods m_β and m_ρ are masked in our example (selected items are encircled, masked items are not).

6.2.4 Pre-treatments

Pre-treatments enable the use of super-blends and embedded selections. They should not be confused with local inheritance. The logical relationships involved with super-blends and embedded selections normally escape to the topological mechanism of local inheritance : the pre-treatments defined below allow these relationships to be taken into account and to be uniformly treated at run-time by local inheritance.

a) Embedded selections

A chroma (in a c-graph or in a p-graph) may belong to several selections. (An example will be given afterwards for the *Person* color graph.) By construction, one of these selections encompasses the other ones. Hence a distinction between the **outer** selection and the **inner** one(s). When such a topology is encountered, the user is not required to specify testing methods at the ephemere nodes of the inner selections. Let's consider the outer selection : because the testing methods attached to its ephemere node are valid in all its basic nodes, these methods are also valid for the inner selections (at their ephemere nodes). These attachments can be done automatically in a pre-treatment phase.

b) Super-blends

Super-blends exist only in p-graphs. Designed for cognitive reasons, a super-blend is a facility meant to symmetricaly express composite transitions and items valid for all states that can be formally materialized by the conjunctions of the basic nodes of selections made along different dimensions (a regular p-chroma may also be part of each conjunction too) : the involved constraints need not to be artificially separated into a clause and an attachment p-chroma. (An example in figure 13 is the super-blend 15 to symmetricaly express the *long-lived* transition -and possibly method- in the *Person* c-graph.) Super-blends are optional : depending on his/her sense of symmetry or any other reason, a programmer may use them or not. Transitions and items defined in a super-blend may be transformed into equivalent constrained transitions and items attached to a p-chroma : the result of ANDing the p-chroma and the clause should be equal to the super-blend. As for embedded selections, this can be done automatically in a pre-treatment phase. This pre-treatment provides uniformity in the rest of the article ; yet, an actual implementation may well manage super-blends in a different way.

6.2.5 Examples

Let's mention a few examples, the first ones being taken from the *Person* color graph.

a) *Person* example

Next figure shows how methods can be attached to the *Person* color graph.

The three *have-birthday* transitions may be implemented by one micro-method (by default, a pre-method) in the *age* pigment : the micro-method is common to each possible alternatives (*child*, *teenager*, *adult*) and is retrieved by inheritance in each case. A *boy* or a *girl* (or any fully initialized instance) inherits the *have-birthday* transition and uses the same method for its implementation.

Conversely, the unique *expected-lifespan* transition is implemented by two micro-methods (by default, pre-methods): one in the *male* pigment, one in the *female* pigment.

Bedtime is basically like *expected-lifespan* : the transition is factorized, but the methods are specific. Yet, instead of specifying one method for each case, one was attached —for demonstration purpose— to the ephemere node *age* : it thus acts as a default. It is masked in *child* and *teenager*, and is inherited in *adult*.

---

[37] Note, however, that a systematic masking in the order of dimensions cannot be guaranteed in all cases : in our example, it is satisfied if m_χ does not exist : m_β then masks m_μ ; and m_μ masks m_ρ. If m_χ exists, then -due to the interleaving of m_μ betwen the two occurrences of m_χ, only m_χ can mask m_μ (the first dimension masks the second one), but m_μ cannot, at the same time, masks m_χ (the second dimension cannot mask the third one).

The *long-lived [3]* transition may be implemented by one micro-method (by default, a pre-method) attached either to the pigment *age* or to the pigment *sex*. In any case, the method is constrained by the other pigment. When finding it in the *age* p-ancestor-tree or in the *sex* one, it is thus declared as being found in both p-ancestor-trees. The success condition is hence satisfied. Being duplicated, the method is retained once after duplicate elimination : the selected occurrence is formally the *sex* one (leftmost dimension). Note this solution works for all initialized instances, not simply for the *boy* and *girl* cases. Note also the same result should be obtained if the *long-lived* transition and method are formally attached to the super-blend 15 (cf. figure 13) : among other approaches, this is obviously the case if a pre-treatment is applied (cf. §6.2.4.b).

The micro-methods for testing testing *male* and *female* are placed in the ephemere node 3. Note that one of them is not strictly necessary. It may be deduced from the other (because basic nodes should form a partition of the ephemere node).

Similarly, the micro-methods for testing *child*, *teenager* and *adult* are placed in the ephemere node 7 (only two of them are strictly necessary). Note that the user may omit the specification of methods for the inner selections, i.e. for respectively testing *child* (and/or *teenager*) in ephemere node 8, and *teenager* (and/or *adult*) in ephemere node 9 (cf. §6.2.4.a). These methods are obtained from ephemere node 7 (outer selection). Consider, for example, node 8 : because the *child* testing method of node 7 is inherited both in nodes 10 and 11, it is valid in node 8.

The micro-methods for testing *child*, *teenager* and *adult* are placed in the ephemere node 7. Note that one of them is not strictly necessary. It may be deduced from the other two (basic nodes should form a partition of the ephemere node).

Other methods may be deduced by the system. These are constant methods *sex* (in nodes 2 and 3) ; *age* (in nodes 6 and 7) ; *child*, *teenager*, *adult*, *boy* and *girl* (in nodes 10 to 14). These are also *boy* and *girl* AND methods for any fully initialized instance.

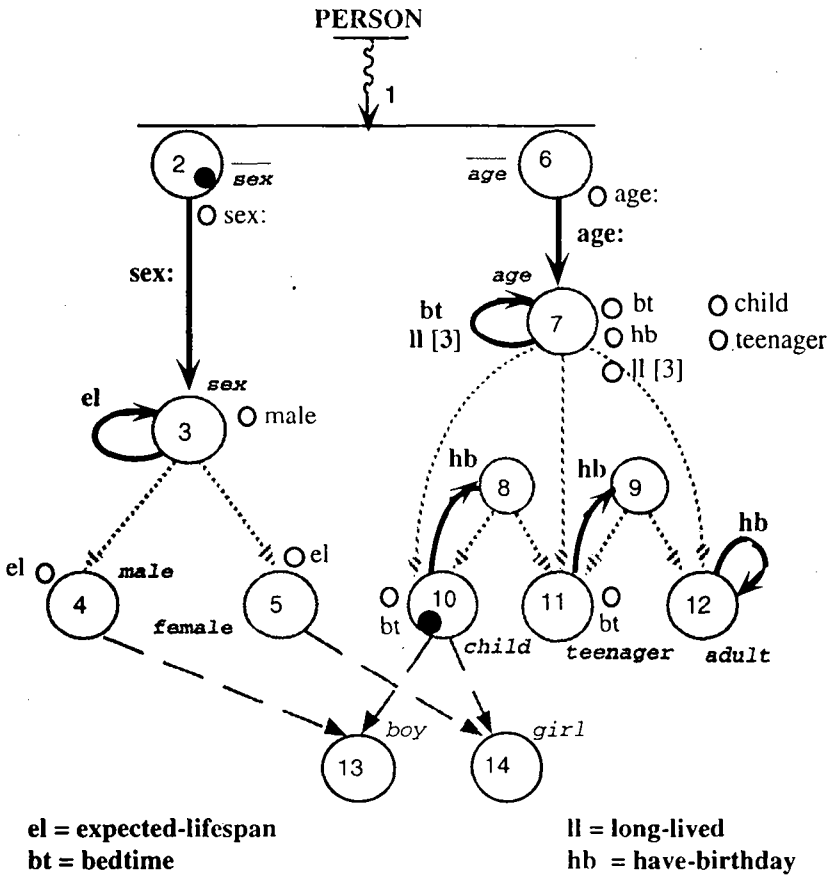


Figure 64.

Using a dimension diagram, next figure abstracts —in terms of dimensions— the search of methods for a fully initialized *Person* instance. The two dimensions exhibited by *Person* are represented by two lines. Methods are depicted by labelled bubbles. They are shown along one or both of these lines depending on the p-chroma to which they are attached in the *Person* p-graph, i.e. depending on the dimension(s) being involved. Let's suppose a *have-birthday* message is sent to the considered instance. Because the *have-birthday* transition is attached to a pigment of the *age* dimension, the search for a *have-birthday* method is done only along the rightmost line (*age* dimension). On the opposite, an *expected-lifespan* message implies a search uniquely along the leftmost line (first dimension). For answering a *long-lived* message, an adequate method is searched along both lines (i.e. both dimensions) from left to right : the same method being found

twice, duplicate elimination yields one occurrence of the *long-lived* method. Note that a slightly different figure would be displayed for a non initialized or a artially initialized instance, a number of methods being different.

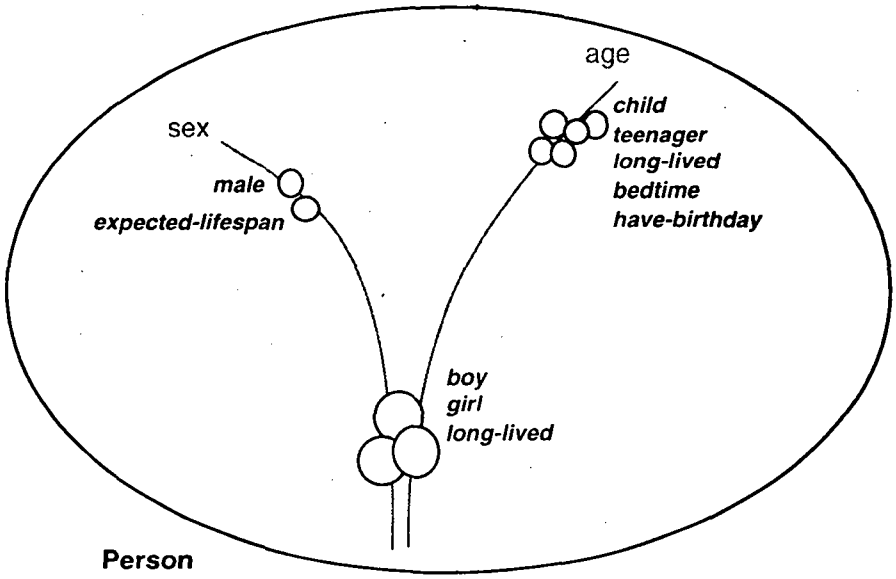


Figure 65.

b) Other examples

An example of micro-method combination is given by an *homothety* operation acting on a given figure. In a *Circle* class, an *homothety* micro-method will be attached to the *radius* pigment, and one to the *center* pigment. According to the above proposal, these two micro-methods will be automatically combined using PROGN. This yields the correct result.

An example of AND combination for evaluating conditions appears in subsection 7.1.3.

7. HIERARCHICAL IMPLEMENTATION

In this section, we expand LOCAL inheritance rules for IMPLEMENTATIONS to CLASS inheritance rules for IMPLEMENTATIONS. This is done progressively : first, we consider superclasses only ; then, we add a local description ; finally, we extend our proposal to a whole hierarchy of classes.

7.1 CONSIDERING SUPERCLASSES ONLY

In this subsection, we consider a class which is a pure combination of its superclasses (no increment attached to it : no extra dimension ; no masking items).

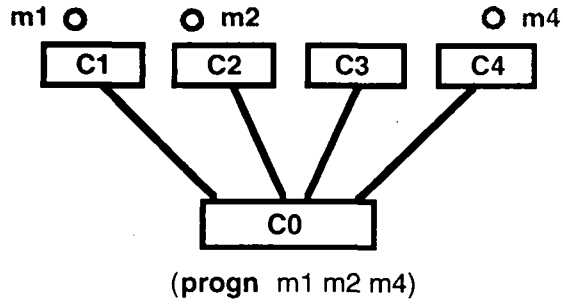
7.1.1 Rules

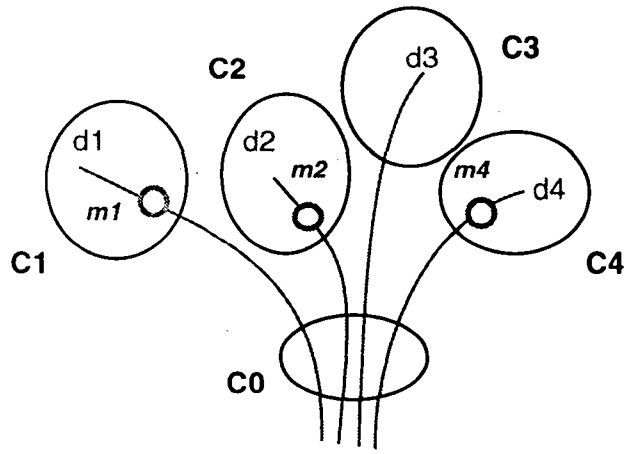
Given a combined class p-graph, CLASS inheritance for TRANSITIONS was obtained by generalizing the LOCAL inheritance rules for TRANSITIONS : it was based on clearing all its p-subgraphs from their transitions and searching these transitions in superclasses (i.e. direct ancestor classes) dimension by dimension. Considered superclasses were the involved ones. The same approach is considered here for implementation items.

Ignoring erroneous cases, our LOCAL inheritance rules for IMPLEMENTATIONS state that
(a) dimensions are ordered according to the order stated in the decomposition construct (given by their first pigment) ;
(b) items are searched dimension per dimension, the considered dimensions being the involved ones ;
(c) concerning methods, the involved dimensions are defined by the the transitions they implement ;
(d) concerning memory representations, all dimensions are involved if not abstract ;
(e) a single item is selected by dimension, the most specialized one (masking effect) ;
(f) if a same item is selected for several dimensions (duplicated item), only its first occurrence is retained ;
(g) all retained items are combined in the order of dimensions (default) ;
(h) the concatenation is used for combining the retained memory representations (default) ;
(i) the AND combination is used for combining the micro-methods retained for the testing transitions (default) ;
(j) the PROGN combination is used for combining the other retained micro-methods (default) ;

If one dimension is exhibited per superclass, then the above rules translate into :
(1) superclasses are ordered according to the list of superclasses ;
(2) items are searched class by class, the considered classes being the involved ones ;
(3) concerning methods, the involved classes are defined by the the transitions they implement ;
(4) concerning memory representations, all classes are involved if not abstract ;
(5) a single item is selected by class, the most specialized one (masking effect) ;
(6) if a same item is selected for several superclasses (duplicated item), only its first occurrence is retained ;
(7) all retained items are combined in the order of superclasses (default) ;
(8) the superclasses memory representations are combined using a concatenation (by default) ;
(9) the superclasses micro-methods for testing transitions are combined using a AND combination (by default) ;
(10) the other superclasses micro-methods are combined using a PROGN (by default) unless a masking effect is sought.

Next two figures illustrate this with a class *C0* having four superclasses (*C1*, *C2*, *C3* and *C4* in this order), each superclass exhibiting a different dimension. *C0* is pure combination of its superclasses. An instance of this combined class is considered. In a given state of this instance, a *m* transition is found in each superclass except *C3* : involved superclasses are thus *C1*, *C2* and *C4*. A method (pre-method) *m* is found in each of these classes. These three methods are noted m_1 in *C1*, m_2 in *C2*, m_4 in *C4*. The (default) combined method is thus (***progn*** m_1 m_2 m_4). (Arguments are omitted.) The first figure shows the class hierarchy and the methods attached to each class. The second figure expresses how methods are specified vs. the dimensions (dimension diagram).

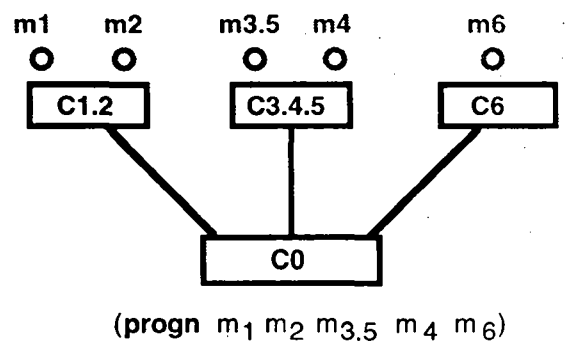


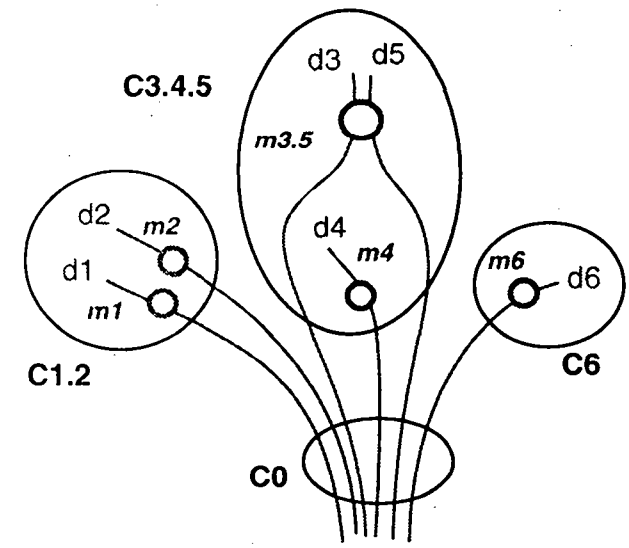


Figures 66 & 67.

A class may in fact exhibit several dimensions. If this occurs, the dimensions are primarily sorted by the order of superclasses in the list of superclasses, and secondarily by the order of dimensions in superclasses exhibiting several dimensions. The above rules are modified accordingly, the main principle still being to proceed dimension by dimension. Note that one item may now correspond to several dimensions.

Next two figures illustrate that with a class *C0* having three superclasses : *C1.2* which provides two dimensions, *C3.4.5* which provides three, and *C6* which provides only one. *C0* is pure combination of its superclasses. As above, let's consider an instance of *C0* in a certain state. To answer a given message *m*, *C1.2* is supposed to provide one (pre)method for each of its two dimensions (be m_1 and m_2 these two micro-methods) ; *C3.4.5*, one that combines its first and third dimensions (be $m_{3.5}$ this method) and one for the second dimension (be m_4 this method); *C6* provides one (m_6). The (default) combined method is thus *(progn* m_1 m_2 $m_{3.5}$ m_4 m_6*)*. (Arguments are omitted.) In this example, all dimensions (and classes) are involved. The first figure shows the class hierarchy and the methods attached to each class ; the second figure brings precision in showing how methods are specified vs. the dimensions (dimension diagram).





Figures 68 & 69.

7.1.2 Modelling the combined method

Let's suppose, as initially done above, that one dimension is stored per superclass. For any given reachable state, say (X Y ...), and any given acceptable message*m* in this state, at most one pair of micro-methods is selected per superclass by the above algorithm. The next figure models that. Superclasses are listed in the order of their declaration : each one determines a level in the figure. X, X',... (resp. Y, Y',...) are instance states of the first (resp. second,...) superclass.

The class inheritance rules for transitions determine in these conditions what transitions will a priori be triggered. This determines in turn the next state (X' Y'...) or, more exactly, the group (X' Y'...) in which the next state will be found. Classes for which no transition exists are not involved (their level may well be removed from the figure).

The inheritance rules exposed above determine what pair of micro-methods implements each transition.

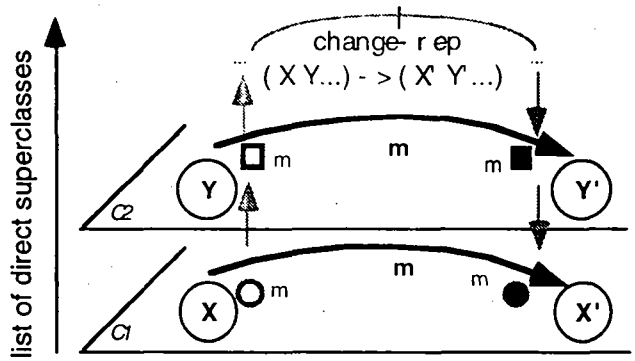


Figure 70.

In these conditions, the general form of a combined method is thus (**progn** m_{p1} m_{p2} ... m_{pn} m_{qn}... m_{q2} m_{q1}). (Arguments are omitted.) Here, m_{pi} and m_{qi} are respectively the pre- method and the post-method in the i^{th} superclass. Some m_{pi} and/or m_{qi} may be absent.

If some superclasses hold more than one dimension, the form of the result is basically the same. Unless not involved, each superclass is associated to a group of one or several levels. Each level may correspond to one or several (involved) dimensions. Next figure models that for an example inspired from the one described by figures 68 & 69 (post-methods have beeen added).

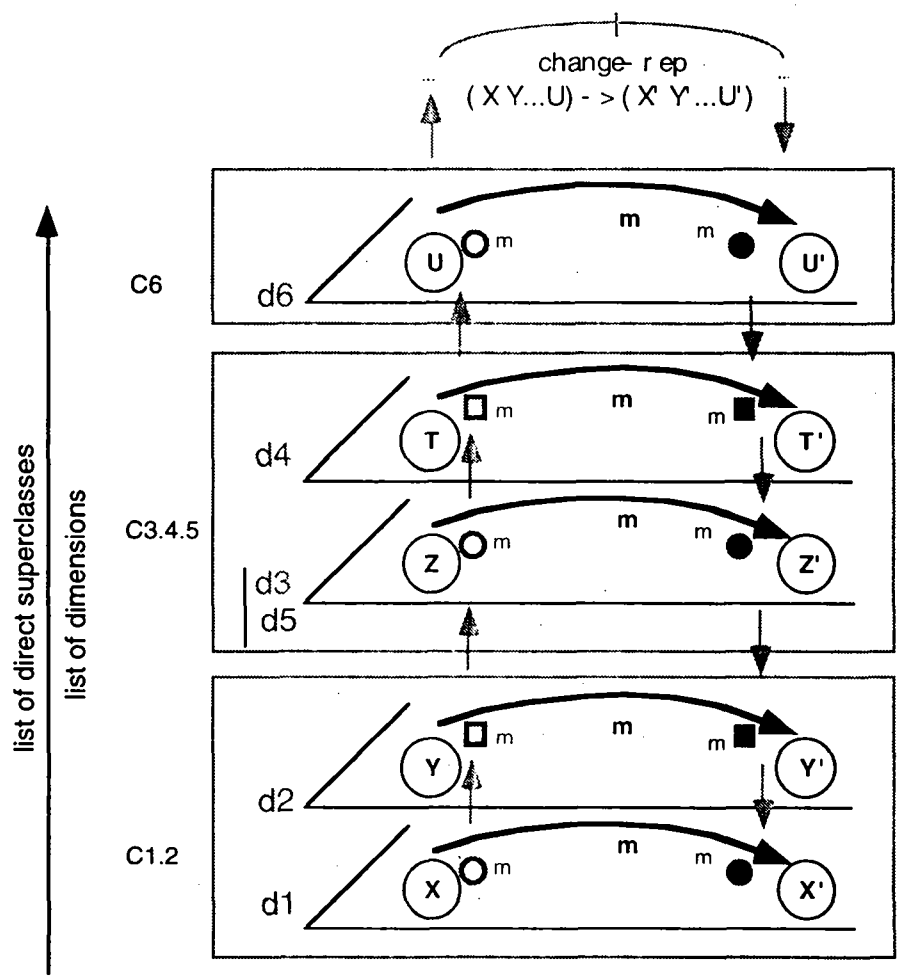


Figure 71.

7.1.3 Example

Let's consider again the *STQ* class with superclasses *Stack* and *Queue* in this order (see figures 33, 34 or 36). Be *(b y)* the considered state, i.e. the one that combines *not empty* in superclass *Stack* and *not empty* in superclass *Queue*.

If the message under exam is *push:*, then a single transition is a priori involved, the one that outcomes from node *b* in *Stack*. Consequently, a single superclass is involved, the *Stack* one. The considered transition being circular, the next state is *(b y)*. Let's suppose the implementation of superclass *Stack* is the one described in §6.1.2.b.1 (no *push:* pre-method). Then the search algorithm discovers but the *push:* post-method attached to node *b*. No combination is to be done : the transition and the method to be run coincide with those of *Stack*.

If the message under exam is *pop*, then two transitions are a priori involved, the *(b c)* one in superclass *Stack* and the *(y z)* one in superclass *Queue*. Yet, given the definition of *STQ*, the *pop* transition is a *masking* one : the *Queue* contribution is thus rejected. Hence, a single superclass is in fact involved, still the *Stack* one. The next state is a substate of *(c y)*. If the *Stack* class is implemented as done in §6.1.2.b.1, then a *pop* pre-method is found in node *b* of *Stack*. The remark done for *push:* is valid in this case too.

If the message to consider is *top*, a similar analysis can be made. The difference is about the next state : it is *(b y)*.

If the message to consider is *empty*, then two transitions are a priori involved : the first one is *(b b)* in *Stack* ; the second one is *(y y)* in *Queue* . Hence, (a priori) both superclasses are involved. The next state is *(b y)*. Here, methods are constant methods : the *Stack* one delivers a *false* answer ; the *Queue* one, too. The result -which ANDs both- is thus a constant : it is *false*.

Etc.

These results are shown below for *push:*, *pop* and *empty*.

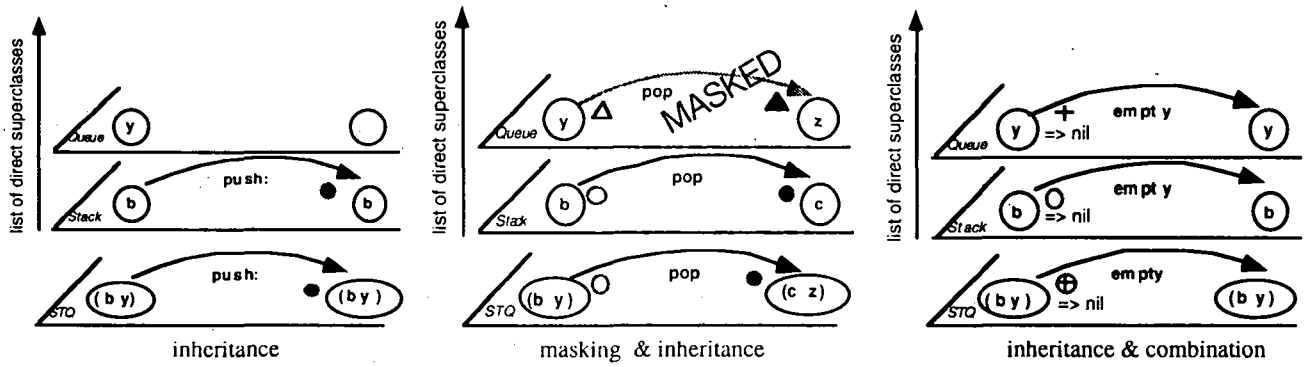


Figure 72.

7.2 TAKING INTO ACCOUNT A LOCAL DESCRIPTION

We now consider that the root class *C0* provides a local description. As for the inheritance of TRANSITIONS, we distinguish a local description that does not add extra dimensions (noted *VSubC0*) and one that does (noted *VSuperC0*) :

a) in the former case, the root class may well propose refined items : each one will mask one or several items in superclasses ;

b) in the latter case, the increment can be taken into account as a virtual superclass, the first one in the list of superclasses. As for transitions (cf. §5.2.1.b.4), CLASS inheritance rules for IMPLEMENTATIONS can thus be extended simply to take into account an increment.

Of course, both cases may be mingled in a same class *C0*.

In the next subsections, we first develop the virtual superclass modelling before illustrating it with the *Bag* example. An example with a mixin (*Bounded-Stack*) will also be shown. The *Person* example is used for illustrating a class exhibiting refined items only.

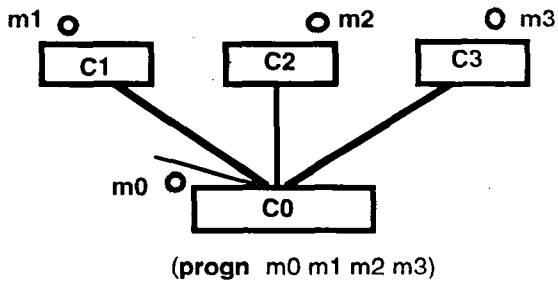
7.2.1 Rule

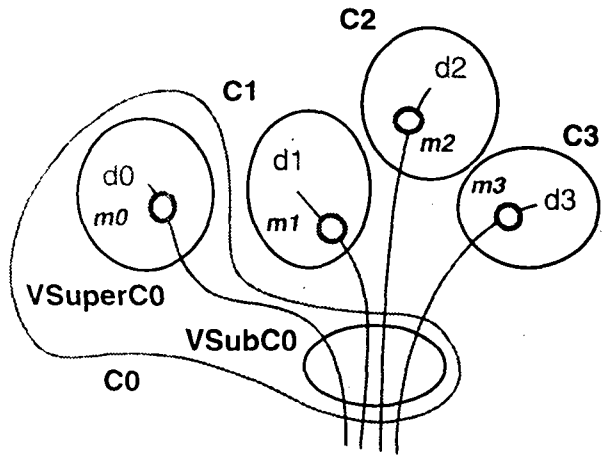
As for transitions, the increment (possibly along several dimensions) is considered before all superclasses. We keep considering it as a virtual superclass (noted *VsuperC0*). Let's consider the various cases in order of increasing complexity.

a) One dimension per superclass & No refined items.

In the simplest case, only one dimension is supported per superclass (virtual one included) and no refined items are provided by *C0*. Let's consider an instance of *C0* in a given state. If m_{pi} and m_{qi} are respectively the pre-method and post-method available in a superclass*Ci* for handling a message *m* in a certain state (with *i* =0 for the virtual superclass), the general form of the combined method is *(**progn** m_{p0} m_{p1} m_{p2} ... m_{pn} m_{qn}... m_{q2} m_{q1} m_{q0})*. Note that methods m_{pi} and m_{qi} may be used in other states than the one under consideration.

Next two figures illustrates this for a class *C0* that inherits from three actual superclasses. Considering only pre-methods, the (default) combined method is *(**progn** m0 m1 m2 m3)*. (Arguments are omitted.) The first figure shows the class hierarchy and the methods attached to each class (the virtual superclass is represented as an oblique line) ; the second figure shows how methods are specified vs. the dimensions (dimension diagram). *VSubC0* represents the refined items (none in this case).





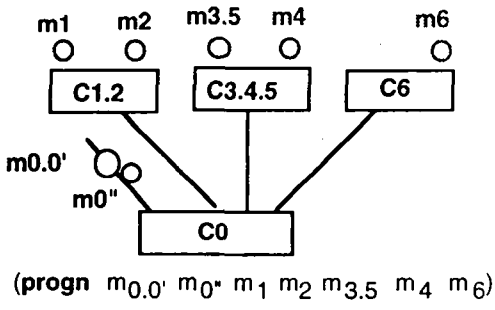
Figures 73 & 74.

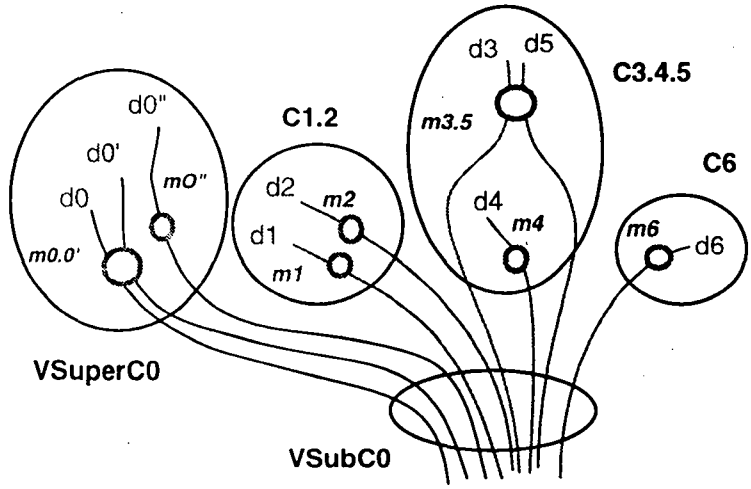
Note that classes *C1*, *C2* and *C3* should themselves be decomposed into a virtual superclass and a virtual subclass. The figure was simplified for these classes since they propose no refined items. For each one, the virtual subclas is not represented ; and the virtual superclass contour is confounded with the class contour itself.

b) Possibly several dimensions per superclass & No refined items.

If some superclasses (virtual one included) exhibit more than one dimension, the form of the result is not very different from above. In a superclass exhibiting several dimensions, a method may now be defined along two or more of its dimensions.

The example shown in the next two figures is derived from the one depicted in figures 68 and 69. *CO* inherits from *C1.2*, *C3.4.5* and *C6*. *C1.2* (resp. *C3.4.5* and C_6) is a class that exhibits two (resp. 3 and 1) dimensions ; the virtual superclass itself is supposed to exhibit three dimensions, noted *d0*, *d0'* and *d0"*. Pre-methods $m_{3.5}$ and $m_{0.0'}$ illustrate the case of (pre-)methods defined along several dimensions in a superclass, possibly the virtual one.



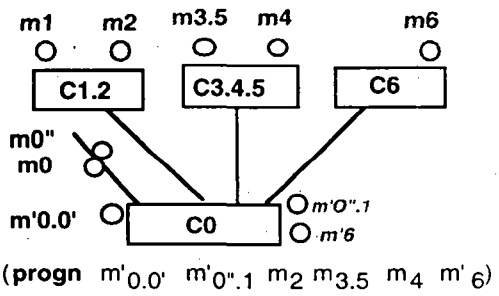


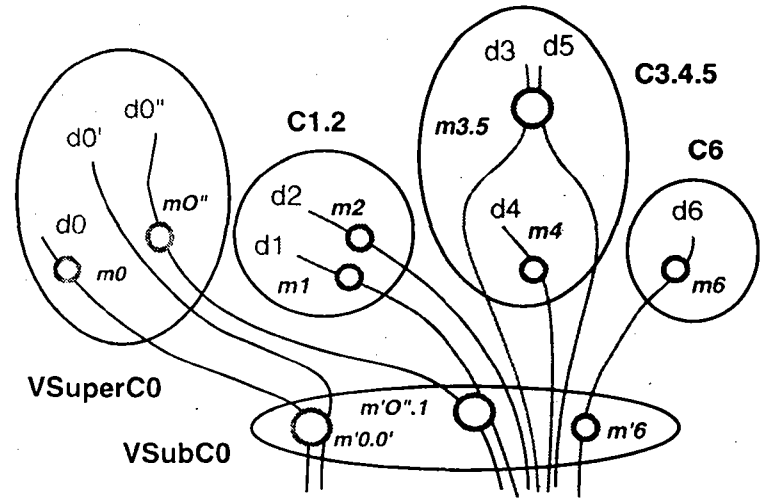
Figures 75 & 76.

This modelling may seem satisfactory. Combined items, i.e. items defined along several dimensions (like $m_{3.5}$ and $m_{0.0'}$) are not considered as being refined items. However, suppose a method *m0* was existing, then *m0.0'* would get the status of refined item and would thus be part of *VSubC0* (*m0* would itself be part of *VSubC0*). A similar remark can be made about *m3.5* vs. *VSubC3.4.5* considering a method *m3* in *C3.4.5*. Next subdivision takes care of that and proposes a better modelling[38].

c) Possibly several dimensions per superclass & Refined items.

Items defined in superclasses may be refined in *C0* : they may be defined along one or several dimensions inherited from one or several superclasses (including the virtual one). These items may thus mask items defined in actual superclasses. In the next two figures, such (pre-)methods are noted *m'*... Considering an instance in the same state as in above figure, m'_6 masks m_6 in *C6* ; $m'_{0".1}$ masks $m_{0"}$ in the virtual superclass and m_1 in *C1.2* ; *m'0.0'* (formely *m0.0'*) masks *m0*. This illustrates that a line represents a refinement path vs. the associated dimension. (Since masked methods were not shown in our previous dimension diagrams, *m0*, *m0"*, m_1 and m_6 should normally not appear in figure 78.) Although not shown in the figure, items defined in *C3.4.5* should also be separated into *VSuperC3.4.5* (*m4*) and *VSubC3.4.5* (*m3.5*).



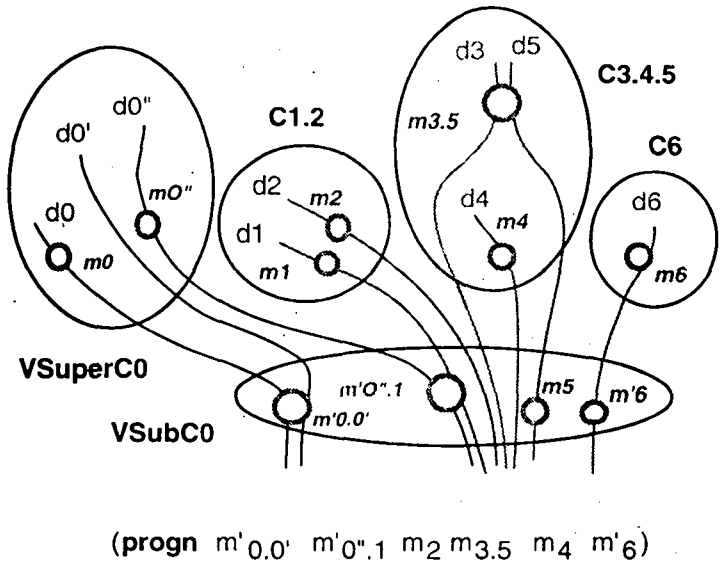


Figures 77 & 78.

38 Distinguishing a combined virtual level doesn't help : *m3.5* would be associated to it ; if *m3* is added in *C3.4.5*, *m3.5* nothing changes for *m3.5* ; but, if *m3* is now move to a superclass of *C3.4.5*. *m3.5* would clearly become a refined item. This internal change of status for an external reason is not satisfactory.

Because each involved dimension should be satisfied once and only once, it is not possible to add methods in an anarchic way : a method like *m'2.5* or *m'4.5* in *C0* (*VSubC0*) would cause an error since *m3.5* would also be selected to satisfy *d3* : as a result, *d5* would be satisfied twice.

Adding a refined method *m5* in *C0* (*VSubCo*) normally produces an error (dimension *d5* is satisfied twice due to the selection of *m3.5* by *d3*) unless we relax class inheritance with a supplementary rule : if an item (here, *m3.5*) satisfies several dimensions in a superclass of *C0*, then *C0* inherits this item if it is not masked by an item of *C0* satisfying at least the same dimensions (here, the concurrent item is *m5*). Strictly speaking, this rule means *m3.5* is also inherited in case both *m3* and *m5* are defined in *C0*. (One can imagine that *m3.5* possibly calls *m3* and *m5* when sent to a *C0* instance, and similar methods in *C3.4.5* when sent to an instance of this superclass.) In the rest of the paper, this rule is termed "**prevalence of combined items**". It can be understood as a generalization of a basic idea underlying the "p-graph local search and combination" rule (which itself is a generalization of the "c-graph local search" rule) : get the most specialized items first. This basic idea is to be retained, but its initial implementation —which basically keeps but the first encountered item— is too narrow : designed in case of a single p-graph, it needs to be adapted to a hierarchy of color graphs.



Figures 79.

In case of conflicts between two prevalent combined item, two simple rules apply :
(1) the most specialized one is selected vs. their common dimension(s) ;
(2) the order of dimensions is taken into account if the previous criteria does not suffice.

7.2.2 Examples

a) The *Person* example

a.1) Modelling

This first example does not add extra dimensions (no virtual superclass to consider), yet refined items are defined (the virtual subclass exists). To display the virtual superclass, the alternative representation proposed for figure 25 is used here : the superclasses c-graphs are normally shown ; the *Person* virtual subclass (the increment) is drawn like the *Person* p-graph, but without any transition/condition names except for *boy*, *girl* and *long-lived*. This kind of presentation is termed an **augmented delta p-graph**, or simply a **delta p-graph**.

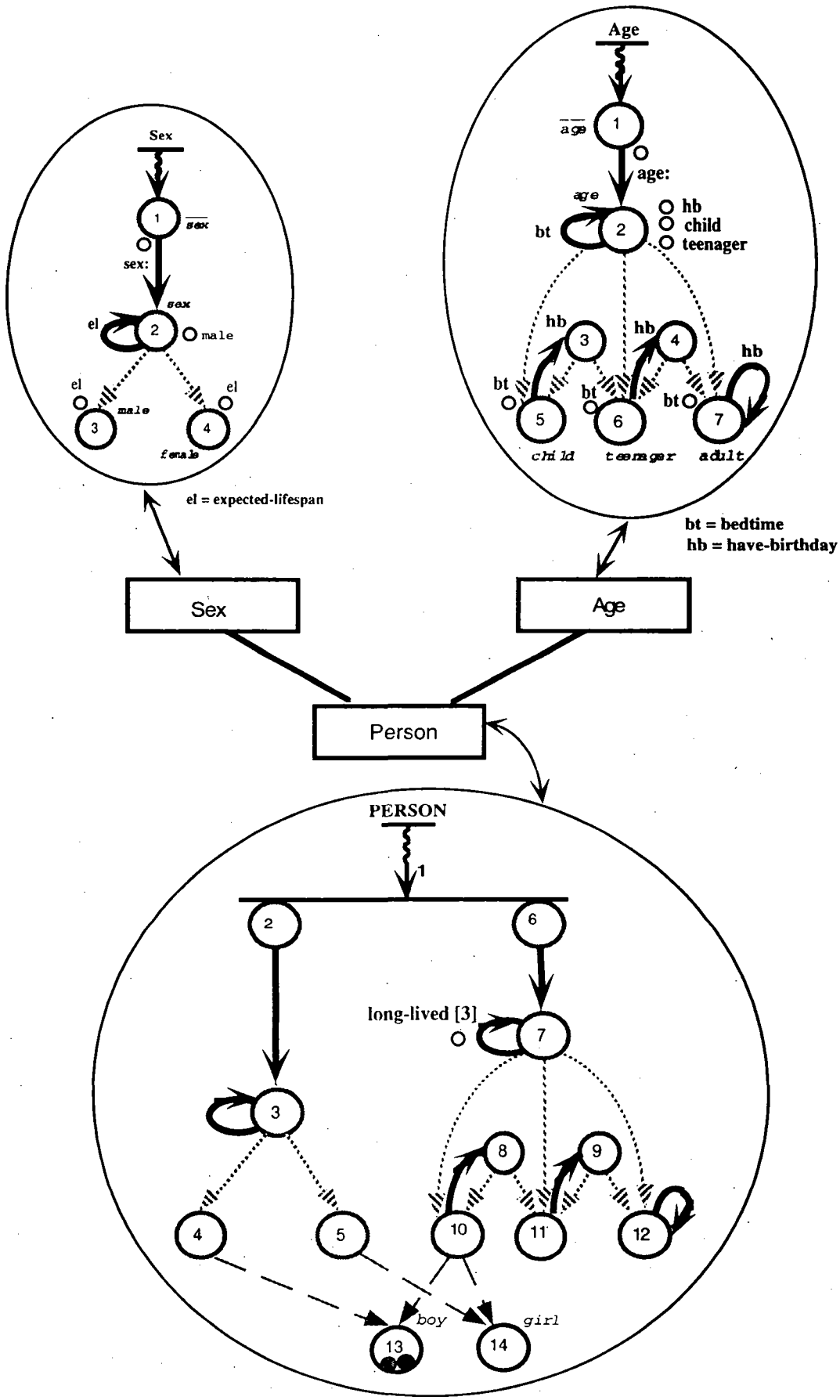


Figure 80.

Suppose the instance current state is *boy* (see next figure). When receiving a *have-birthday* message, the search is first made locally (*age* dimension) : no micro-method is found that way. The search continues in the involved superclass, the

Age one : a *have-birthday* method is found in node 2. (This method is in fact valid for any *Person* instance the *age* of which is initialized [be it *child*, *teenager* or *adult*].) If the *long-lived* message is sent to the instance, the local search provides a method (attached to pigment 7 of *Person*)...

a.2) Dimension diagram

Next figure expresses, in terms of dimensions, how the search proceeds when searching a method for a totally initialized *Person* instance : the *sex* (resp. *age*) dimension is represented by a line traversing the *Sex* (resp. *Age*) and *Person* classes. A method is first searched in *Person*, then in one of its superclasses depending on the involved dimension(s).

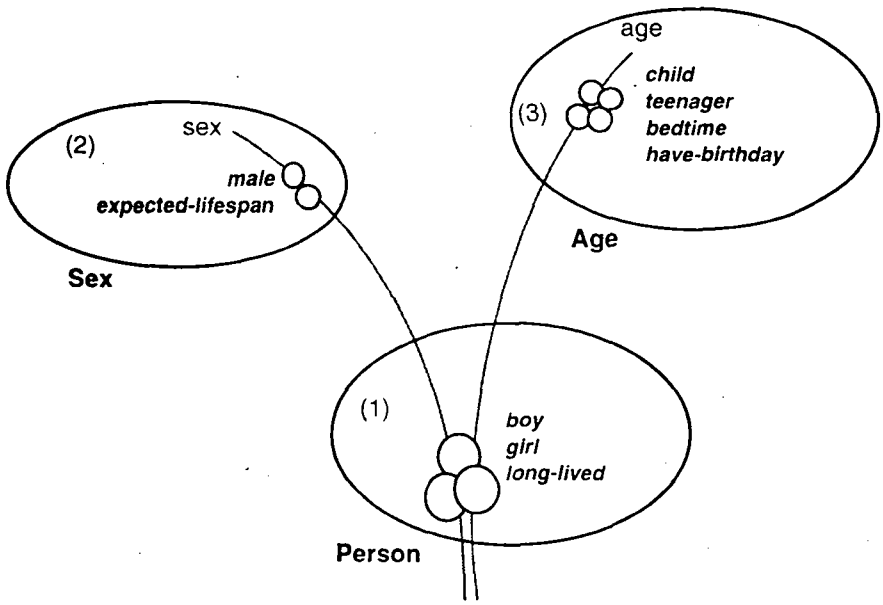


Figure 81.

b) The *Bag* example

b.1) Description

Our second example is about the *Bag* class, a subclass of *Object*. The *Bag* color graph was shown in figure 39 (§5.2.1.b.2). It results from the composition of a local increment with the *Object* color graph.

Next figure shows the augmented *Object* color graph (it exhibits one *print* method) and the augmented *Bag* p-graph (the implementation defined for the local increment is, for instance, similar to the one of *Stack* in section 6 : it is characterized by no *elements* cell in *empty* and a post-method *put:* in *not empty*). A specific *print* method has been attached to *a* and another one to *b*, both with clause α.

b.2) Modelling

Given the above definitions, *Bag* can be understood as comprising :
— a virtual superclass (termed *Bag-Virtual-Superclass* or *BVS* for short) since the increment depicted above adds a new dimension. *BVS* defines transitions, memory representations and methods (*put:*, *get*, *empty*) ;
— a virtual subclass (say *Bag-Virtual-Subclass or Bvs for short*) since *Bag* refines the *print* method of *Object*.

Figures 83 and 84 show apart the *BVS* c-graph and the *Bag-Virtual-Subclass* (in delta form) as they result from the above definition. The latter inherits the *print* transition from *Object*. The two *print* methods are associated to blends $a.\alpha$ and $b.\alpha$ to emphasize they do belong to the subclass of *BVS* and *Object*, and not to *BVS* (no *print* transition is defined in *BVS*).

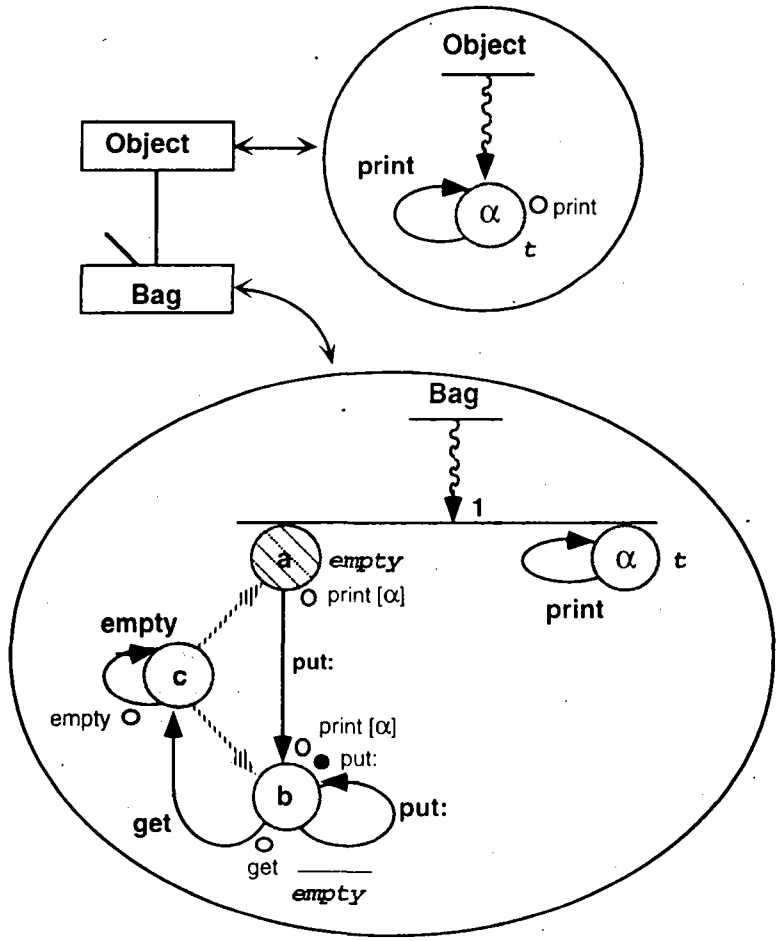
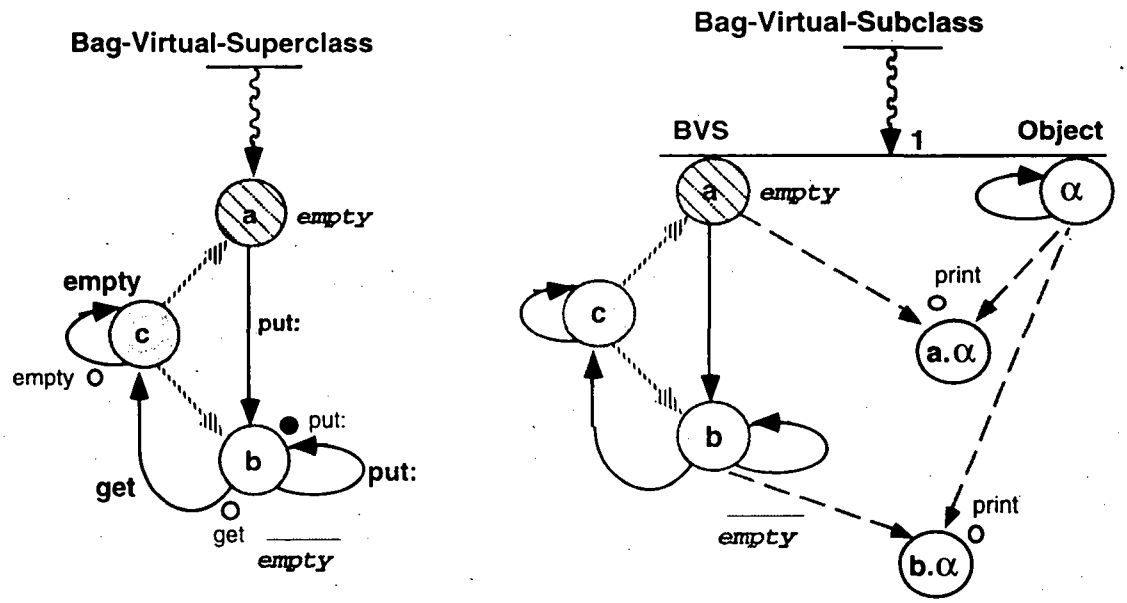


Figure 82.



Figures 83 & 84.

b.3) Dimension diagram

Suppose we consider a *Bag* instance in a given state. When an items is searched for this state, the delta p-graph is first examined. Then, due to class inheritance, inspection is done in the color graphs of its superclasses which are -in order- the superclass*BVS* (i.e. the local increment) and then the superclass *Object*.

In terms of dimensions, the *Bag* class adds an extra dimension (say *bag*) to the dimension inherited from *Object* (say *object*). The *bag* dimension is described by *BVS*. When searching is done, an item associated with both dimensions is privileged ; if not found, the *bag* and *object* dimensions are inspected in that order : the search is done first in *Bag* , then -if unsuccessful- in the adequate superclass (*BVS* or *Object*). An opportunity is thus given to mask an item defined in *Object*, here the *print* method, either by attaching an item to *Bag* for both the *bag* and *object* dimensions, or uniquely for the *object* dimension. Next figure shows the associated dimension diagram : the *object* (resp. *bag*) dimension is represented by a line traversing the *Object* and *Bvs* classes (resp. *BVS* and *Bvs* classes). The figure is drawn for a *Bag* instance in a *not empty* state ; in case of an *empty* state, the *get* method would not be present in *BVS*.

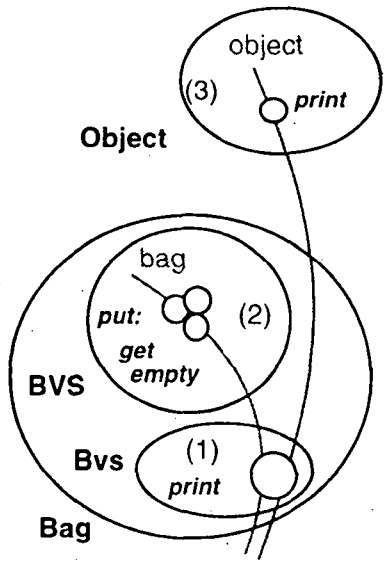


Figure 85.

b.4) Searching methods

Let's consider a *Bag* instance in a given state and suppose we are looking for methods.
— Because the *put:*, *get* and *empty* transitions are defined for the *bag* dimension, the corresponding methods are searched along this same dimension. They are found in *BVS* (i.e. in the local increment).
— The *print* transition being defined for the *object* dimension, a *print* method should satisfy this dimension : in *Bag*, it can thus be attached to both the *object* and *bag* dimensions and/or to the the *object* dimension ; if not found in *Bag*, the *print* method of *Object* is inherited (default method). Considering a *Bag* instance, two methods are found in*Bvs* : one corresponds to an *empty* instance ; one, to a *not empty* instance. Both mask the default *print* method of *Object*. In the dimension diagram (just above), the method found for a *not empty* instance is represented by the bubble inside *Bvs*. It satisfies both dimensions. This method is selected since it satisfies the involved dimension (here, *object*) once and only once. It also satisfies the *bag* dimension which is not involved. This is not an error : a non involved dimension may be **employed**, i.e. satisfied by a selected item : the unique constraint is to satisfy this extra dimension once and only once (as if it was involved[39]).

Note that these two *print* methods constrained by pigment α, and attached to the pigments *a* and *b* of the local description (see figure 82) are equivalent to two methods constrained by pigment *a* and b (these cover all possible cases), and both attached to pigment α. In the dimension diagram (just above), this does not make any difference. In the example, no *print* method is attached in *Bag* to the sole α pigment. This could be the case if the memory representation in pigments *a* and *b* were the same (attaching the method to both pigments *c* and α is equivalent).

---

39 One way to understand that consists in developing an equivalent modelling. As done in §5.2.1.b.2/footnote 28, the abstract dimension *object* is re-integrated into the *Bag* class. Similarly to the *print* transition, the default *print* method, initially stored in *Object*, is also reintegrated into *Bag*. For example, it is attached to pigment α and is given the non void clause : *[c]*. Alternatively, it can be attached to pigment *c* and be given the non void clause *[α]* ; or, it can be attached —without any clause— to a blend resulting from the conjunction of α and *c*. In any case, there are two involved dimensions : *object* and *bag*. Thus each one should be satisfied once and only once, in particular the *bag* dimension.

c) The *Bounded-Stack* example

The addition of a mixin can be considered as the addition of an increment. This is coherent with the inheritance point of view : we saw that the mixin superclass should be placed before the base superclass (cf. §5.2.2.b.5). This increment is added but in one state : compared to the behaviour of a *Stack* instance, the behaviour of a *Bounded-Stack* instance is altered in color 2 only (*not empty*). This state has substates *full* (pigment β) and *not full* (pigment α). When searching a method in (a 2) or (b 2), the search starts locally, then in the mixin, then in the base color graph. Due to the implicit masking, items found in the mixin color graph are not searched in the base color graph. In this case, the *full* method is found locally ; the other methods are inherited from the *Stack* superclass.

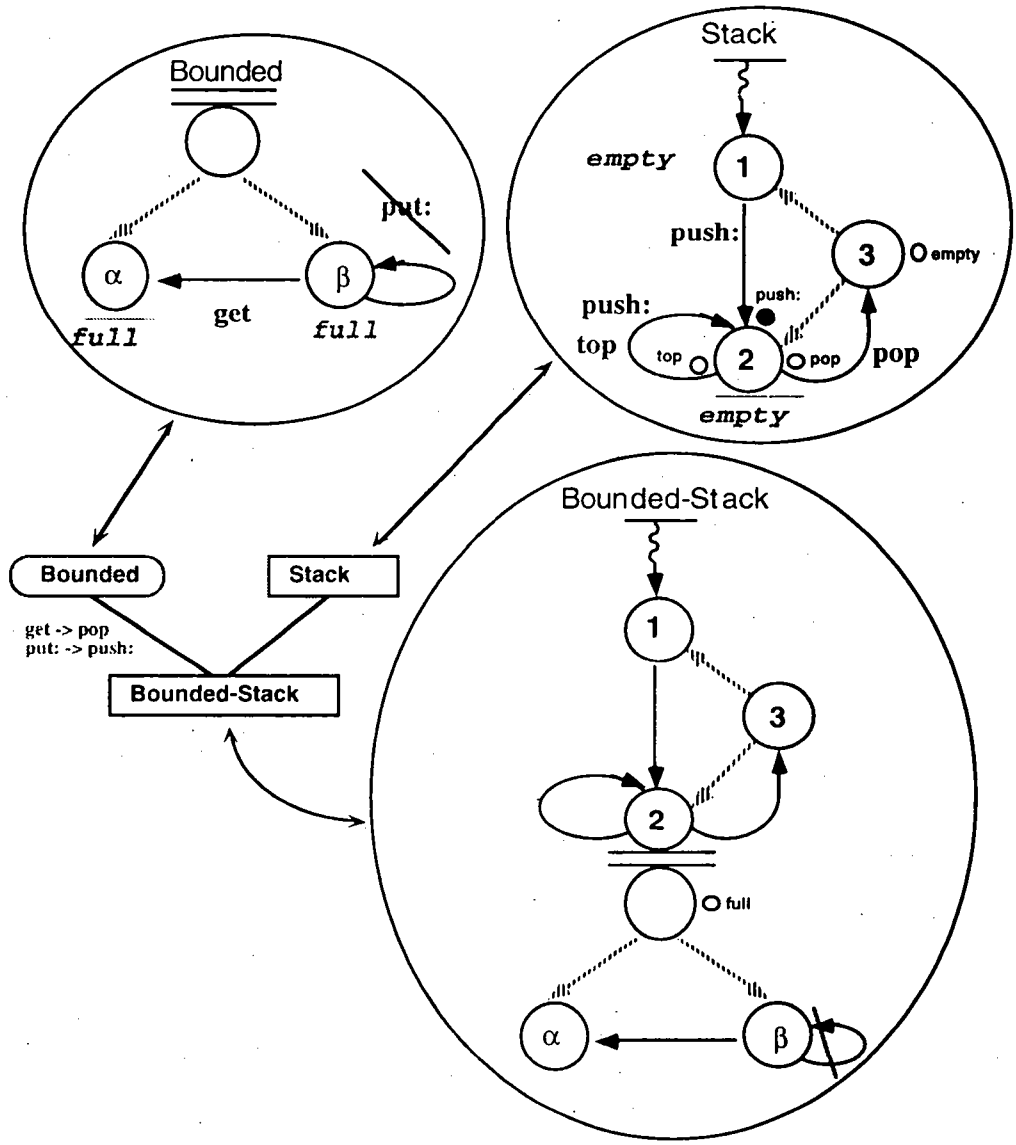


Figure 86.

The above results may be presented in terms of dimensions. Next figure is drawn for a *not empty instance*: In this state, the *Bounded-Stack* instance is described along its two dimensions, the *stack* and *bounded* one : the former (resp. later) is depicted by a line traversing the *Stack* (resp. *Bounded*) and *Bounded-Stack* classes. When *empty* , a *Bounded-Stack* instance is described along the *stack* dimension only.

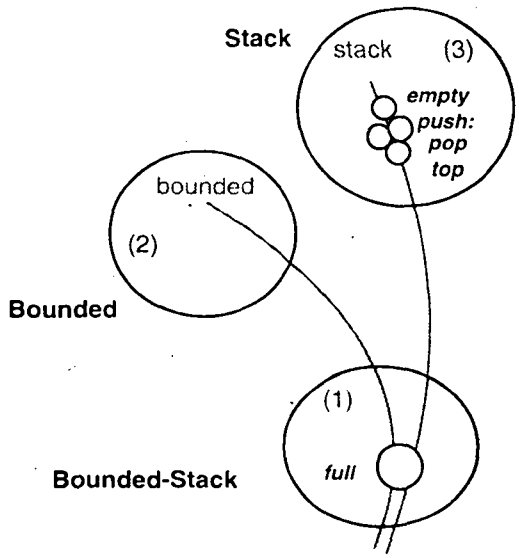


Figure 87.

7.3 CONSIDERING ALL ANCESTORS

Having considered the inheritance relationship of one class with its superclasses, we now come to its generalization : the inheritance relationship of this class with all its ancestors, direct or indirect. Whereas a class and its superclasses form a tree, a class and all its ancestors do not generally constitute a tree but a DAG (Direct Acyclic Graph, or lattice): common ancestor classes exist in the hierarchy as they are devised for sharing common properties (an example is the *Object* class). As first step, we consider a tree structure (next subsection). Then, the DAG structure is considered (subsections afterwards) : first, we sumarize our analysis ; then, we give examples ; finally, we propose a linearization algorithm[40] after having discussed its role.

7.3.1 Tree Structure

Even if it does not correspond to the more general case, the tree structure is important in practice since parts of a given hierarchy may well be trees (if not the whole hierarchy). A tree structure corresponds to single inheritance.

a) Inheritance rule

The whole process of searching an item (for example, a micro-method) in a given instance generalizes by recursion what was done previously when considering a class having but direct ancestors (i.e. superclasses) :
- apply the local inheritance rules (more precisely, the "p-graph local search and combination" rule) in the corresponding color graph ;
- if an item has not been found locally for a given dimension, take the one in the corresponding superclass (same dimension), by applying the same local search rule in the superclass color graph ; repeat this same process recursively using the ancestors of this superclass in case of still unsuccessful search. If a dimension remains finally unsatisfied, an error is signalled.

b) Example

As an example, we consider the following figures. They are derived from figures 77 & 78. First one shows a class hierarchy.

---

[40] Given root class *C0*, a linearization algorithm produces a list L_{C0} of all the classes that belongs to the hierarchy H_{C0} of the root class. In CLOS, this list is termed the class precedence list (*cpl*, for short). Implicit but important : a list is ordered and any of its elements appears but once it it. Besides the term linearization, we also use ordering in the same sense.

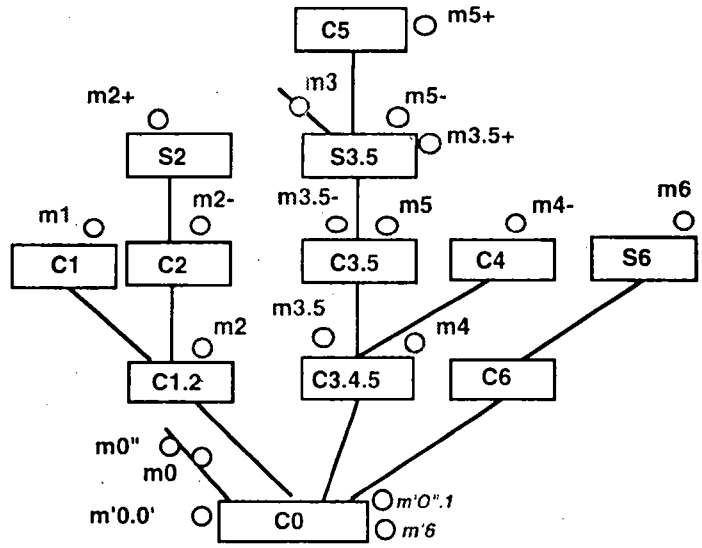
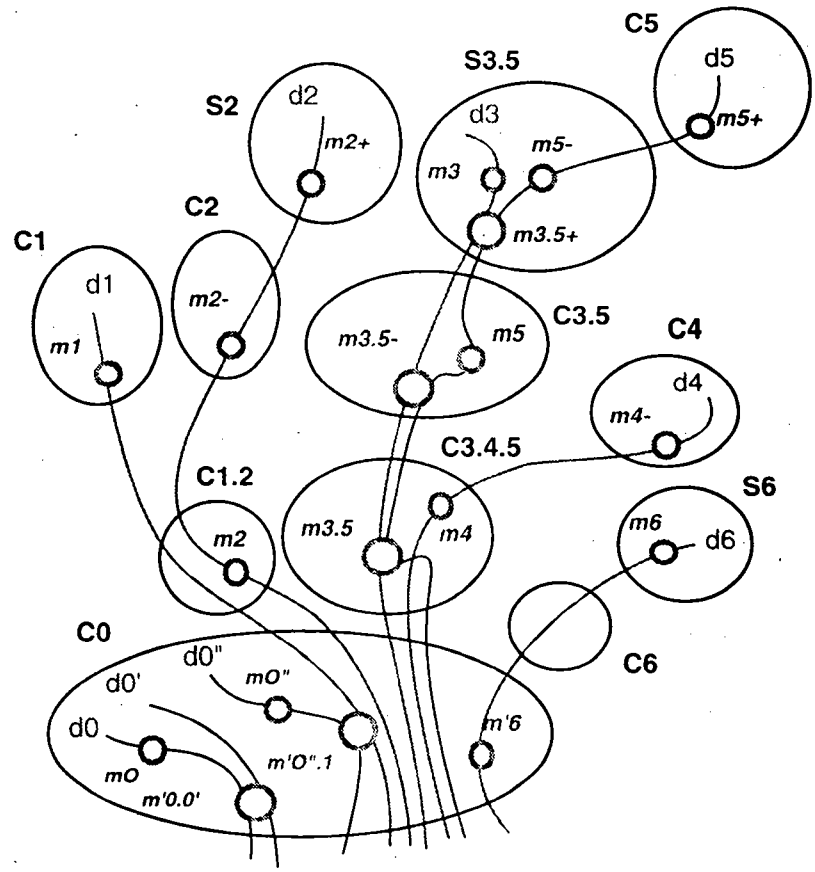


Figure 88.

Next figure shows a dimension diagram for a given state. For simplicity, the parts VSuper and VSub of classes *C0* and *S3.5* are not isolated.



(progn $m'_{0.0'}$ $m'_{0".1}$ m_2 $m_{3.5}$ m_4 m'_6)

Figure 89.

In our example, inheritance yields the same result than for figures 77 & 78.

c) More examples

If we interpret the above figure as the superposition of a number of situations, i.e. if previously selected methods may well be absent, new methods will be selected (unless an error is detected). If these methods are in turn absent, other methods will be used (unless an error is detected). And so on, until no method is avalable.

c.1) running the "p-graph local search and combination" rule

The figure below lists the various possibilities depending on the situation. A situation is a set of cases. One case is to be considered for each set of dimensions as listed below. The above result is obtained in situation (1 1 1 1 1 1). In situation (1 2 3 5 2 2), all conditions cumulate and the result is *(m'0.0' m0" m1 m2+ m3 m5+ m4- m6)*. In a given situation, all or part of the methods above the selected ones may not exist in ancestor classes : the result is unchanged for a *C0* instance.

The table is established without taking into account the "prevalence of combined items" rule (§7.2.1.c). For dimensions *d3* and *d5*, if *m3.5* and *m3.5-* do not exist, *m5* gets selected : since *m3.5+* gets also selected for satisfying *d3*, *d5* gets satisfied twice ; as a consequence, an error is detected. This is why, for these dimensions, line 3 also requires *m5* to not exist. For each dimension or group of dimensions, the last line (labelled "-") mentions the supplementary conditions for having no selected items : usually, this is but the items that were previously selected (i.e. those mentionned in the line just above). For the group *d0* and *d0'*, we have an extra condition : if *m'0.0'* is supposed to not exist, then an error occurs since the dimension *d0'* is not be satisfied ; thus, *m0* should not exist too (or a *m0'* method should exist too). For simplicity purpose, these last lines would not be mentionned in similar tables.

Dimensions	Case	Conditions to cumulate	Methods to combine
d0, d0'	1		$m'_{0.0'}$
	–	~~$m'_{0.0'}$~~ ~~m_0~~	
d0", d1	1		$m'_{0".1}$
	2	~~$m'_{0".1}$~~	$m_{0"}$ m_1
	–	~~$m_{0"}$~~ ~~m_1~~	
d2	1		m_2
	2	~~m_2~~	m_{2-}
	3	~~m_{2-}~~	m_{2+}
	–	~~m_{2+}~~	
d3, d5	1		$m_{3.5}$
	2	~~$m_{3.5}$~~	$m_{3.5-}$
	3	~~$m_{3.5-}$~~ ~~m_5~~	$m_{3.5+}$
	4	~~$m_{3.5+}$~~	m_3 m_{5-}
	5	~~m_{5-}~~	m_3 m_{5+}
	–	~~m_3~~ ~~m_{5+}~~	
d4	1		m_4
	2	~~m_4~~	m_{4-}
	–	~~m_{4-}~~	
d6	1		m'_6
	2	~~m'_6~~	m_6
	–	~~m_6~~	

Figure 90.

Next figure represents the same results using an **invocation sequence diagram** instead of a table. This diagram lists the successive possibilities (selected items) obtained for each dimension or group of dimensions : the conditions to cumulate are now partly explicit (ex. : absence of *m5* on the branch *m3.5-* / *m3.5+*), partly implicit (a possibility is selected if previous ones do not exist). Given our example, the structure we get is a **forest** (i.e. a list of trees). An invocation diagram does not mention classes. As such, it is a bit more abstract than the dimension diagram it derives from.

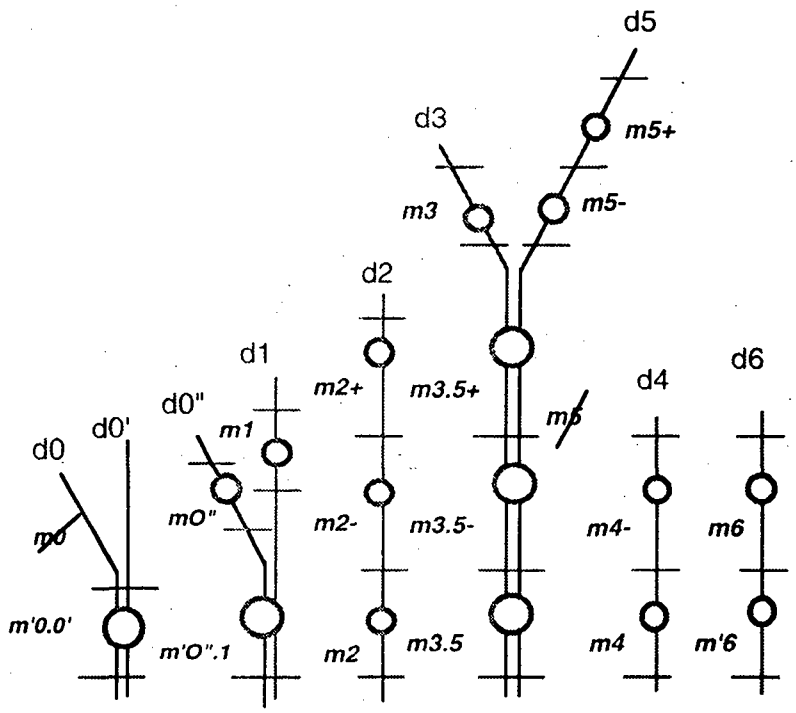


Figure 91.

c.2) taking into account the "prevalence of combined items" rule

The above table was established without taking into account the "prevalence of combined items" rule (§7.2.1.c). Let's now take it. The table is not modified except for the group of dimensions *d3* and *d5*. The presence of *m5* is no longer a source of error. Next figure shows the part of the table that gets modified.

Dimensions	Case	Conditions to cumulate	Methods to combine
d3, d5	1		$m_{3.5}$
	2	~~$m_{3.5}$~~	$m_{3.5-}$
	3	~~$m_{3.5-}$~~	$m_{3.5+}$
	4	~~$m_{3.5+}$~~	m_3 m_5
	5	~~m_5~~	m_3 m_{5-}
	6	~~m_{5-}~~	m_3 m_{5+}

Figure 92.

Next figure shows the new invocation sequence diagram. No explicit condition is mentionned now for dimensions *d3* and *d5* (absence of *m5*) ; the branch *d5* is modified (presence of *m5* before *m5-* and *m5+*).

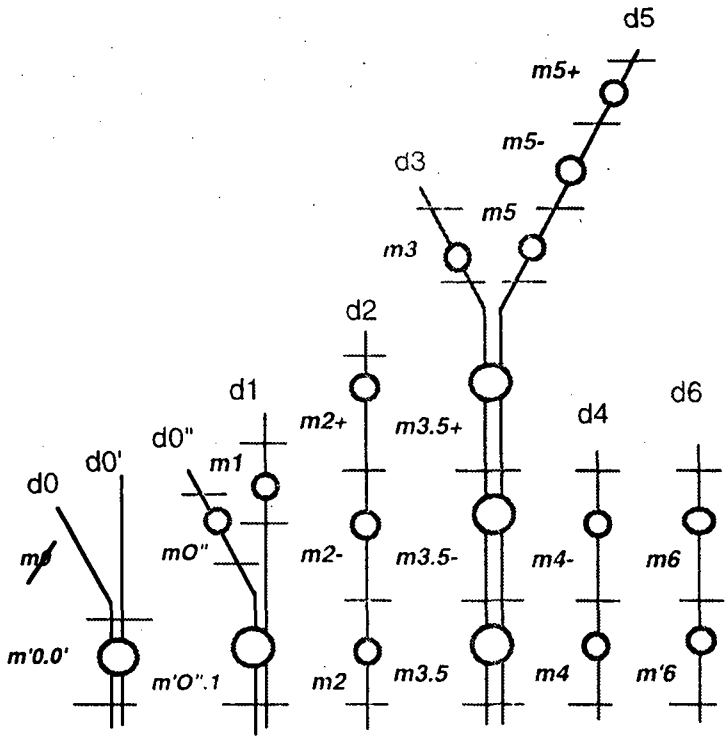


Figure 93.

7.3.2 DAG Structure

Let's now consider the general case : a DAG structure. This subsection abstracts our analysis. The subsections afterwards expands it notably with a detailed example and an especially elegant linearization algorithm.

a) Unicity or multiplicity of inherited dimensions

In our analysis, we distinguish two cases, depending on whether a dimension inherited along two or more paths of classes is duplicated two or more times or is considered as unique (default case). To take an example, it is clear that the identity of a *person* (say, its *social security number*) is unique and cannot be cloned as it could be for a *card number*.

These two opposite ways of handling inherited dimensions may a priori seem irreconcilable. At this point, we have to remember that the concept of dimension relates to the space in which the states of a given class of objects are described : a dimension corresponds to an axis. Hence, the correct way to interpret the unicity or multiplicity of an inherited dimension is the state space : in case of unicity, the dimension in question corresponds to one axis and only one ; in contrast, in case of multiplicity, the axis is duplicated as many times as required.

Note that the choice between unicity and multiplicity is not to be made when specifying a composition (the specification is the same). As shown by the *STQ* example, the choice can be further delayed and be isolated in a subclass.

b) Inheritance rule

b.1) Multiplicity case

Suppose all dimensions inherited along two or more paths of classes are duplicated two or more times : the inheritance structure, initially a DAG, is transformed into a tree. The above rule for a tree structure thus applies.

b.2) Unicity case

For each dimension (axis), a sub-hierarchy of classes is a priori involved. What we do is to **linearize each sub-hierarchy**. These independent linearizations may not agree on the order of common classes. Hence, the possibility of conflicts for items. In this case, we say that the dimensions are **conflicting**.

b.2.1) no conflict

α) organization to consider

When no conflict exists, the organization we get is a list of classes for each dimension. This is generally not a tree since any two dimensions may have several class sublists in common. However, all classes are listed in the same order. The recursive application of the "p-graph local search and combination" rule, as for a tree, yields the correct result. No other computation (ex. : sorting) needs to be done.

Let's take an example. We consider an instance of *C0* in a given state. Three dimensions are supposed to exist : *dx*, *dy* and *dz*. Next figure shows an hypothetic dimension diagram obtained in such a case.

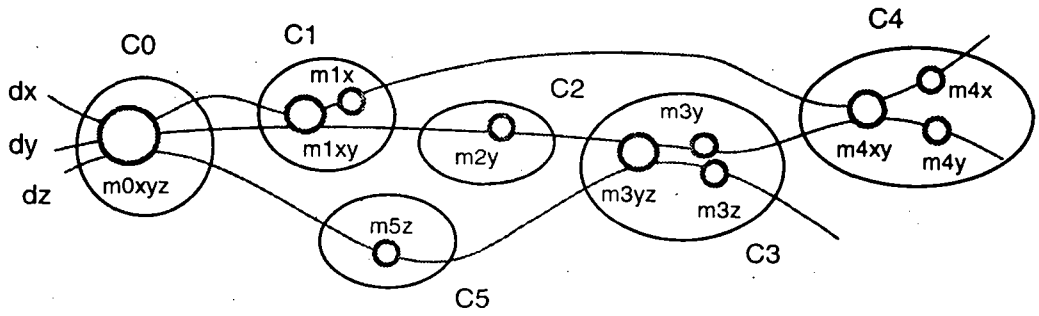


Figure 94.

β) running the "p-graph local search and combination" rule

Recursively running the "p-graph local search and combination" rule on the same dimension diagram first selects *m0xyz*. If *m0xyz* does not exist, we get two methods to combine : *m1xy* and *m5z*. If both *m0xyz* and *m1xy* do not exist, three methods are to combine : *m1x*, *m2y* and *m5z*. If one of these methods does not exist (in addition of *m0xyz* and *m1xy*), an error occurs since one of the other dimensions will be satisfied twice (due to the selection of either *m4xy* or *m3yz*). We thus have to suppose that both *m2y* and *m5z* do not exist (in addition of *m0xyz* and *m1xy*). Given this, we have to combine *m1x* and *m3yz*. Now, if we add a further condition (*m3yz* does not exist), the methods to combine are *m1x*, *m3y* and *m5z*. Etc. The next figure lists all possible selections with the conditions to cumulate.

	Conditions to cumulate	Methods to combine
1		m0xyz
2	~~m0xyz~~	m1xy m5z
3	~~m1xy~~	m1x m2y m5z
4	~~m2y~~ ~~m5z~~	m1x m3yz
5	~~m3yz~~	m1x m3y m3z
6	~~m1x~~ ~~m3y~~	m4xy m3z
7	~~m4xy~~	m4x m4y m3z

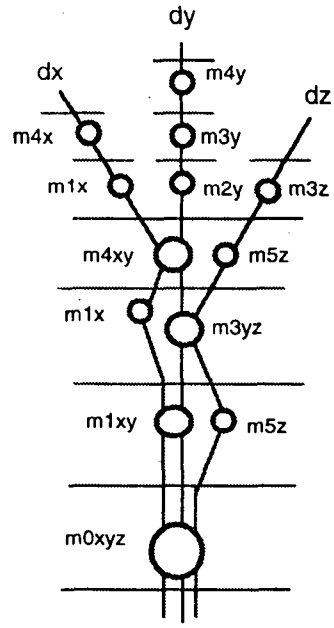
Figure 95.

γ) taking into account the "prevalence of combined items" rule

Let's now recursively run the "p-graph local search and combination" and the "prevalence of combined items" rules on the same dimension diagram. First, *m0xyz* is selected as before. If *m0xyz* does not exist, we get two methods to combine : *m1xy* and *m5z*. If both *m0xyz* and *m1xy* do not exist, *m3yz* prevails on *m2y* and *m5z*, hence two methods to combine : *m1x* and *m3yz*. (Note that *m3yz* prevails on *m4xy* : this one is placed after it for dimension *dy*.) The next

figure lists all possible selections with the conditions to cumulate. (Once in step 5, the three dimensions get independent : results are to be combined according to *dx*, *dy* or *dz*.) The figure afterwards shows the methods to combine in a graphical way using an invocation sequence diagram : note the structure we get is not a tree as obtained in the preceding example.

	Conditions to cumulate			Methods to combine		
1				m0xyz		
2	~~m0xyz~~			m1xy		m5z
3	~~m1xy~~			m1x	m3yz	
4	~~m3yz~~			m4xy	m5z	
5	~~m4xy~~			*	*	*
	dx	1		m1x	-	-
		2	~~m1x~~	m4x	-	-
	dy	1		-	m2y	-
		2	~~m2y~~	-	m3y	-
		3	~~m3y~~	-	m4y	-
	dz	1		-	-	m5z
		2	~~m5z~~	-	-	m3z



Figures 96 & 97.

δ) supposing in addition a "regular" hierarchy

A well chosen example is meant to explore the limits of what can theoretically happen given the adopted hypotheses. In this spirit, the above dimension diagram illustrates the possible coexistence of methods with **interleaving** dimensions : the diagram shows the coexistence of both mi_{xy} ($m1_{xy}$ or $m4_{xy}$) and $m3_{yz}$. Such a coexistence effectively happens in the real world : for instance, in the *STQ* example, the *print* method of *Stack* (*LIFO*, *bag* and *object* dimensions) and the *print* method of *Queue* (*FIFO*, *bag* and *object* dimensions) do induce interleaving once the linearizations are done (see figure 129). In a subsequent subsection, we show that, given our choices, the problem never occurs for memory representations (see 8.3.1). Let's now consider the case where the problem does not exist for methods too. We say that a hierarchy is **regular** vs. its methods named *m*, if all *m* methods of degree≥ K that exist along a same dimension do satisfy the same *K* dimensions[41]. The corresponding property is termed **regularity** : it is most important when considering a systematic combination style. (For more details about regularity, see subsection 8.3.)

In our example, this property means *m3yz* does not exist. The figure below shows the new hierarchy.

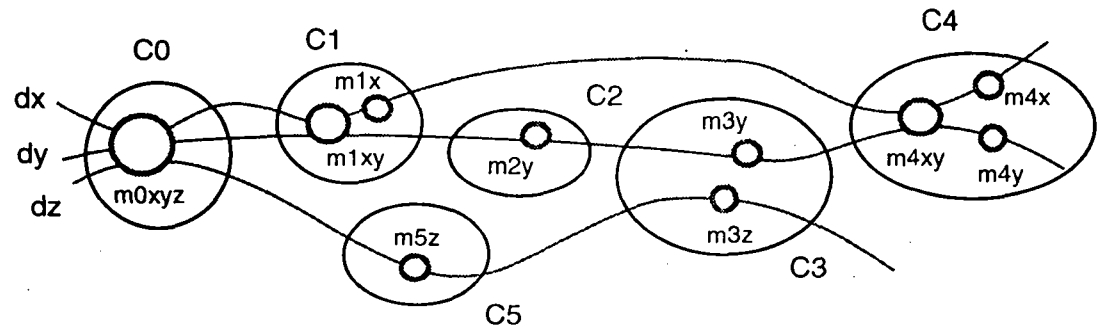


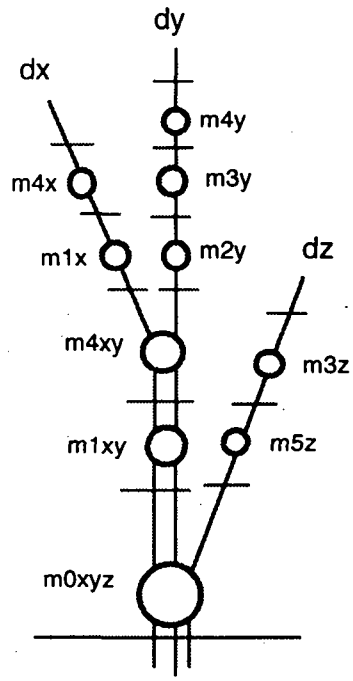
Figure 98.

The effect of this modification on the sequence of combinations that would be chosen depending on the presence or absence of methods is two faceted :
— direct : because *m3yz* disappears, line 4 in the previous table (figure 95) no longer exists in the new table ;
— indirect: because *m3yz* was blocking *m4xy*, other changes occur due to the "prevalence of combined items" rule.

41 a method of degree K satisfies K dimensions (K≥1).

Next figure lists all possible selections with the conditions to cumulate. The figure afterwards shows the same results using an invocation sequence diagram : note the structure we get is now a tree as obtained in the preceding example (cf. fig 93).

	Conditions to cumulate			Methods to combine		
1				m0xyz		
2	~~m0xyz~~			m1xy		*
	dz	1		-		m5z
		2	~~m5z~~	-		m3z
3	~~m1xy~~			m4xy		*
	dz	1		-		m5z
		2	~~m5z~~	-		m3z
4	~~m4xy~~			*	*	*
	dx	1		m1x	-	-
		2	~~m1x~~	m4x	-	-
	dy	1		-	m2y	-
		2	~~m2y~~	-	m3y	-
		3	~~m3y~~	-	m4y	-
	dz	1		-	-	m5z
		2	~~m5z~~	-	-	m3z



Figures 99 & 100.

b.2.2) One or more conflicts

α) organization to consider

The conflict of two or more dimensions is a potential source of ambiguity. Consider, for example, two conflicting dimensions, say *dx* and *dy*. These dimensions order at least two classes differently : for example, *C1* is before *C2* for dimension *dx* while *C2* precedes *C1* for *dy*. Suppose now the existence of methods that are valid for both dimensions and are precisely defined, one in *C1* (say *m1*) and the other in *C2* (say *m2*). Suppose these methods mask all other possible ones. Then, depending on the order of the dimensions *dx* and *dy*, the method to be selected will be different.

Figure 101.

Since dimensions are ordered, a choice is automatically made (for example, *m1* if *dx* is ranked before *dy*). Nevertheless, this conflict solving mechanism is not fully satisfying : it requires from the programmer too much attention to details. Hence, a few questions : is a more regular linearization algorithm possible ? Will its use be a panacea ? If not, what is the best to do ?

Well, we would be interested in a linearization algorithm that systematically orders in the same way all classes common to different dimensions of a same hierarchy. However, it appears to us that a linearization algorithm cannot systematically satisfy both this property (we termed it congruency) and monotonicity/incrementality[42]. Choosing an algorithm (ex. : the LOOPS one[43]) that respects this property will solve a relatively rare problem while compromising monotonicity/incrementality in may be a larger number of cases. Because monotonicity/incrementality is a highly desirable property, we prefer to have some conflicts and to solve them using the ordering of the dimensions. (Note this

42 Monotonicity and incrementality are used in an equivalent way in [Ducournau et alii, 1992] [Ducournau et alii, 1994].

43 The linearization algorithm of LOOPS, which is stable but not always monotonic satisfies the property in question. Demonstration will be given in §7.3.5.c.

solution is similar to what is done in CLOS for sorting the applicable methods having several specializers.) When a conflict exists, the user may be warned on demand by the programming environment.

To summarize, the solution we chose consists in recursively running the "p-graph local search and combination" rule according to the ordering of classes obtained by linearizing the sub-hierarchies of classes associated to each dimension. The default algorithm of linearization is preferably chosen monotonic/incremental : conflicts are then solved by privileging the item ranked first according to the relative order of dimensions (see below). The user is given the opportunity to change the linearization algorithm by MOP (meta-object protocols).

Next figure shows an example. Compared to the previous one, the order of *C4* and *C1* has been inversed for *dx*. The dimensions *dx*, *dy* and *dz* are ranked in this order.

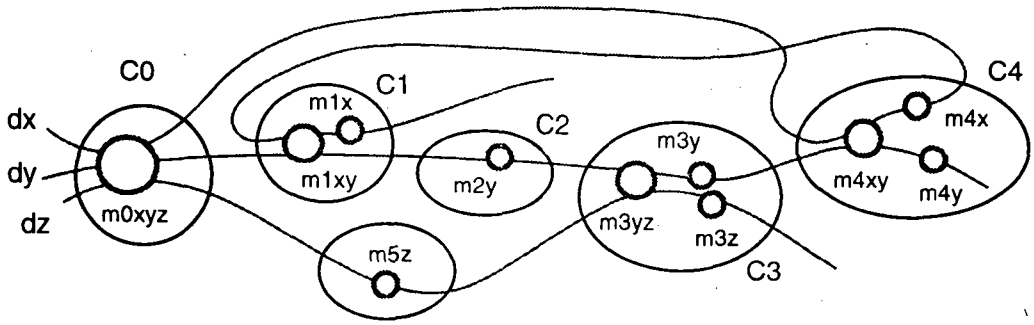


Figure 102.

β) algorithm replacing duplicate elimination

Let's now precisely indicate how conflicts are solved using the relative order of dimensions.

The duplicate elimination phase during the execution of the "p-graph local search and combination" rule usually compares the name of the items a priori retained for each dimension : this rule implements in fact a more general rule saying that each dimension should be satisfied once and only once. In the present situation, we cannot use the usual implementation of this rule.

Here is the more general algorithm : a list of satisfied dimensions is incrementally built up by traversing the list of pre-selected items. When a new candidate item is examined, we throw it away if the dimensions it satisfies are already all satisfied ; when part of the dimensions its satisfies are already satisfied and part of them aren't, an error is detected ; when none of the dimensions it satisfies are already satisfied, this item is kept and the list of satisfied dimensions is updated. When the list of candidate items is exhausted, all dimensions should be satisfied ; otherwise, an error is detected.

γ) running the "p-graph local search and combination" rule (and the new algorithm as well)

When *m0xyz* does not exist, a conflict a priori exists between *m4xy* (found first for *dx*) and *m1xy* (found first for *dy*). Because *dx* is ranked before *dy*, *m4xy* takes precedence over *m1xy*.

Let's be more precise. The list of pre-selected items is (*m4xy m1xy m5z*). To satisfy each dimension once and only once, this list is traversed using the above algorithm. Initially, the list of satisfied dimensions is empty. After having examined *m4xy*, this list is (*dx dy*). The next item is *m1xy* : it satisfies *dx* and *dy* . Thus, it is thrown away. The next candidate item is then *m5z* which satisfies *dz* : it is kept. The list of candidate items is now exhausted. Since all dimensions are satisfied, we get the correct result.

Next figure lists all possible selections with the conditions to cumulate. It is established in case the "prevalence of combined items" rule is not applied. At each step, we suppose that the selected items do not exist. Note that *m4x* poses a problem when *m0xyz* and *m4xy* are supposed not not exist : *m4x* should not exist too (otherwise, dy is satisfied twice due to the selection of m4x and m1xy). Similarly, when *m2y* does not exist, *m5z* should not exist too.

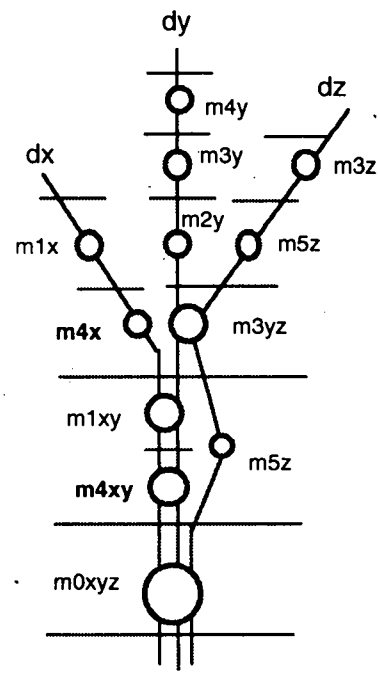
	Conditions to cumulate	Methods to combine
1		m0xyz
2	~~m0xyz~~	**m4xy** m5z
3	~~m4xy~~ ~~m4x~~	m1xy m5z
4	~~m1xy~~	m1x m2y m5z
5	~~m2y~~ ~~m5z~~	m1x m3yz
6	~~m3yz~~	m1x m3y m3z
7	~~m3y~~	m1x m4y m3z

Figure 103.

γ) taking into account the "prevalence of combined items" rule

Next figure shows the same kind of table in case the "prevalence of combined items" rule is applied. Methods to be combined are listed depending on the situation (i.e. on the absence of some other methods). Once in step 5, the three dimensions get independent : results are to be combined according to *dx*, *dy* or *dz*. The figure afterwards shows the same results using an invocation sequence diagram .

			Conditions to cumulate	Methods to combine
1				m0xyz
2			~~m0xyz~~	**m4xy** m5z
3			~~m4xy~~	m1xy m5z
4			~~m1xy~~	m4x m3yz
5			~~m3yz~~	m4x m2y m5z
	dx	1		**m4x**
		2	~~m4x~~	m1x
	dy	1		m2y
		2	~~m2y~~	m3y
		3	~~m3y~~	m4y
	dz	1		m5z
		2	~~m5z~~	m3z



Figures 104 & 105.

δ) supposing in addition a "regular" hierarchy

Let's now suppose that the hierarchies to be considered satisfy the property of regularity. Next figure obeys this constraint (compared to figure 102, *m3yz* has vanished).

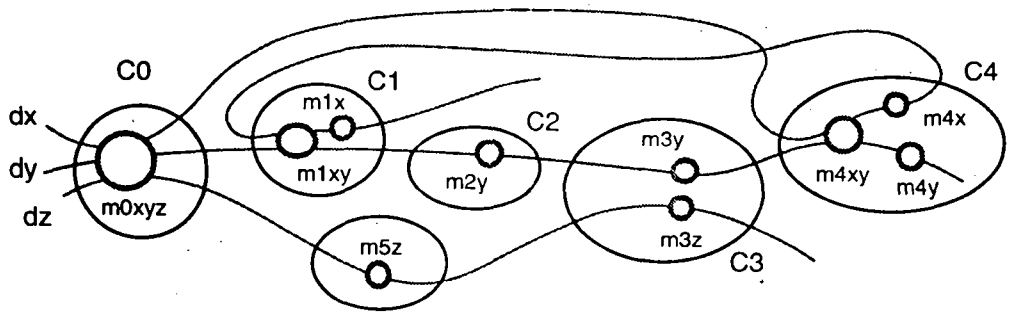
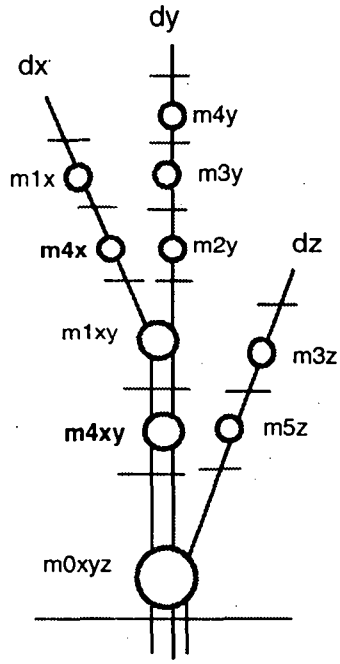


Figure 106.

Next figure lists all possible selections with the conditions to cumulate. Line 4 in the previous table (figure 104) no longer exists in the new table. The figure afterwards shows the methods to combine using an invocation sequence diagram : note the structure we get is a tree. Compared to figures 99 and 100, methods *m1xy* and *m4xy* are found in the opposite order ; idem for methods *m1x* and *m4x* : this is due to the different ordering of dimension *dx* vs.dimension *dy*.

	Conditions to cumulate			Methods to combine		
1				m0xyz		
2	~~m0xyz~~			**m4xy**		*
	dz	1		-		m5z
		2	~~m5z~~	-		m3z
3	~~m4xy~~			m1xy		*
	dz	1		-		m5z
		2	~~m5z~~	-		m3z
4	~~m1xy~~			*	*	*
	dx	1		**m4x**	-	-
		2	~~m1x~~	m1x	-	-
	dy	1		-	m2y	-
		2	~~m2y~~	-	m3y	-
		3	~~m3y~~	-	m4y	-
	dz	1		-	-	m5z
		2	~~m5z~~	-	-	m3z



Figures 107 & 108.

b.3) Conclusion

In this subsection, the selection of valid items in a non-degenerated hierarchy (DAG) was studied (ex. : selection of methods valid vs. a given message sent to an instance in a certain state). These items are basically obtained by recursively running the "p-ancestor-tree local search" rule until found.
Supposing that the selected items do not exist yields a new set of items to be combined. We studied the succession of results obtained doing so. The structure we get may be complex. However, it systematically simplifies into a tree when the following conditions hold :
— the hierarchy is regular ;
— the "prevalence of combined items" rule is observed ;
— the "unique satisfaction of each dimension" rule is observed (in place of the more specialized "duplicate elimination" rule).

7.3.3 Examples

This subsection illustrates the double analysis we just made. A same hierarchy is considered. Be *C0* its root class ; and *d*, a dimension which is multiply inherited in *C0*. Our two examples differ only by the way they handle this dimension. In the first example, all occurrences of *d* are considered to be different (duplication) : all happens as if *C0* was inheriting different dimensions from distinct superclasses. In the second example, the occurrences of *d* are considered to be the same (unification) : this dimension exists only once at the *C0* level.

a) STQ : common stuff

Our two examples consider a same hierarchy, the *STQ* hierarchy (the *STQ* example was introduced in subsection 5.2.1.).

a.1) The *STQ* hierarchy

Next figure shows the inheritance graph of *STQ*. *STQ* inherits from *Stack* and *Queue* ; and these two inherit from *Bag* ; *Bag* inherits from *Object*.

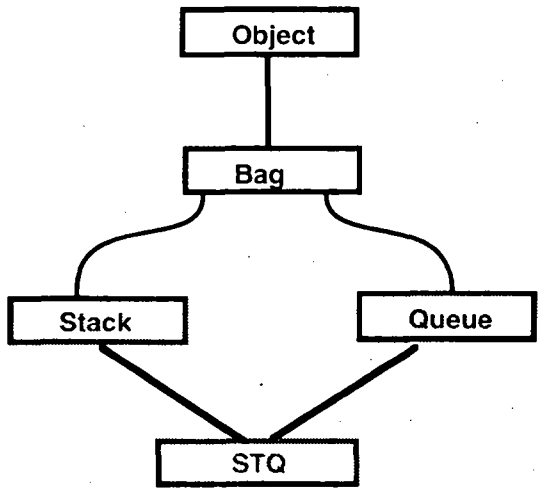


Figure 109.

Object features one dimension (noted *object*) ; *Bag* has two dimensions : one inherited from *Object* and one local (noted *bag*) ; *Stack* and *Queue* have three dimensions : two inherited (from *Object* and *Bag*) and one local (respectively *LIFO* and *FIFO*) ; *STQ* inherits all these dimensions and does not introduce a new one. The figure shows the hierarchy of *STQ* in the **natural ordering of specialization** (placing *Queue* on the left and *Stack* on the right would appear unnatural since *Stack* is before *Queue* in the superclass list of *STQ*). Given a class, each local description that introduces a new dimension is materialized as a virtual superclass (ranked before all regular superclasses). Except for *Object*, the oblique line represents both the dimension and the virtual superclass ; concerning *Object*, it represents but the *object* dimension (*Object* introduces this dimension by itself : no virtual superclass is needed.)

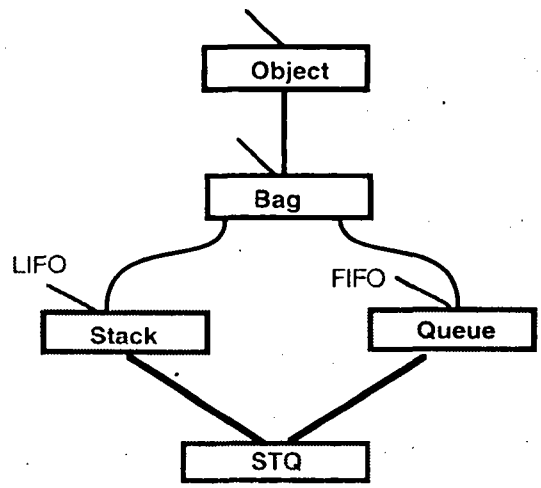


Figure 110.

a.2) The *print* methods : notation

STQ inherits from *Object* the *print* transition. Let's focus on how this transition can be implemented in the *STQ* hierarchy. Next figure shows, for a given state, various *print* methods that may be defined. (We may assume that the user desires prettier and prettier results when informations on the instance get more and more precise, and thus chooses to specify many methods.) A notation is used to name these methods conveniently.

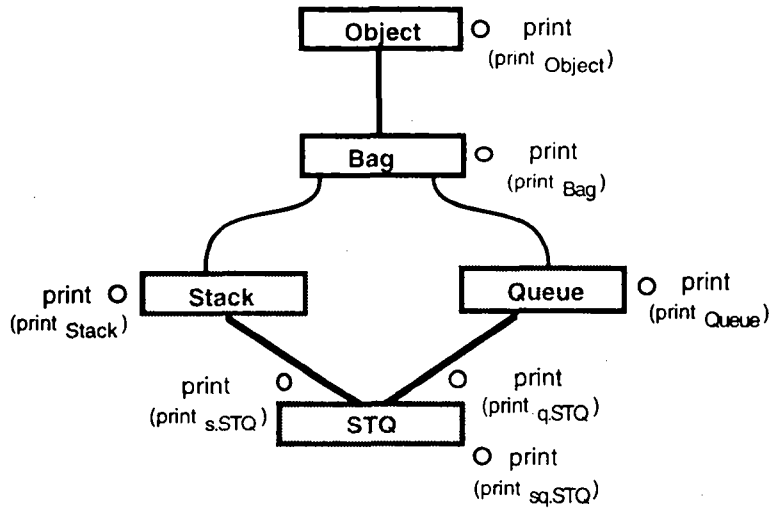


Figure 111.

In each class, the method name is *print*. This name is the first one indicated in the figure. The second one, under parentheses, specifies how each method can be identified in a unique manner in the text below. In *Bag*, *Stack* and *Queue*, the method name is indexed by the class name since the method is unique.

In *STQ*, an other index is used since three methods exist : this index specifies what dimension(s) of an instance are concerned. The *s* index denotes the first three dimensions of a *STQ* instance : *object*, *bag* and *LIFO* ones (globally termed the *stack* "dimension"). The *q* index is used for the three other dimensions, i.e. the *object*, *bag* and *FIFO* ones (globally termed the *queue* "dimension").

As a matter of fact, this same index will also be used when a method is not normally applied. This would be, for example, the case for $print_{Stack}$ when restricted to the *stack* "dimension" of a *STQ* instance instead of the *stack* and *queue* "dimensions" (whole instance). In *bag* (resp. *object*), the dimensions under consideration being *object* and *bag* (resp. *object*), this means the *print* method is further restricted to these sole dimensions : for example, $print_{s.Bag}$ is to be applied only to the second (*bag*) and third (*object*) dimensions of the *STQ* instance.

b) Hypothesis 1 : duplication

b.1) duplication of the *Bag* and *Object* dimensions

In *STQ*, the *object* and *bag* dimensions are inherited along two different paths (say *s* and *q*). In this subdivision, by hypothesis, each occurrence of the *bag* or *object* dimensions is considered to be distinct from the other occurrences. Next figure illustrates that. The *s* and *q* paths are made distinct on their whole length : the subpath *Bag/Object* is duplicated. A net effect of this is thus to face a tree instead of a DAG. In this respect, this example is in direct line with the previous section. (This motivates its study before the default case which effectively examplifies a DAG structure.)

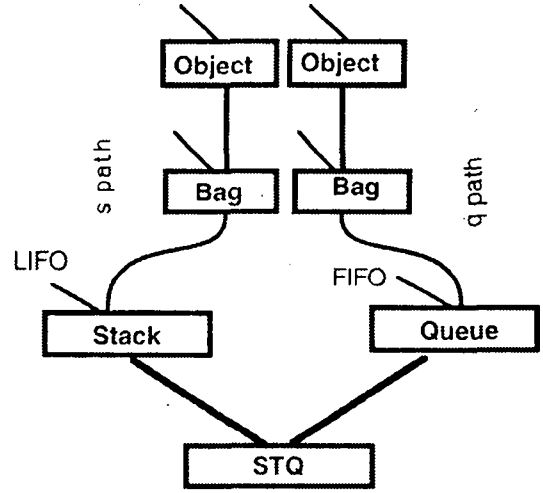


Figure 112.

b.2) interpretation in the state space

In this example, the state space has 6 dimensions (axes) : *LIFO*, *FIFO*, two *bag* dimensions (say *bag1* for *bag [Stack]* and *bag2* for *bag [Queue]*) and two *object* dimensions (say *object1* for *object [Stack]* and *object2* for *object [Queue]*). Next figure tends to illustrate this : a given state of an *STQ* instance (say stq_n) is projected onto the six axes (each axis is normally orthogonal to the other ones).

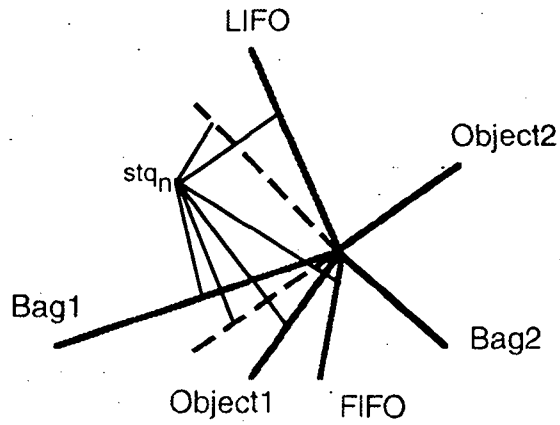


Figure 113.

b.3) impact on the p-graph

Representing a cartesian space having 6 dimensions on a plane is kind of awkward (cf. preceding figure). Fortunately, our formalism is better at that. Next figure shows the corresponding *STQ* p-graph.

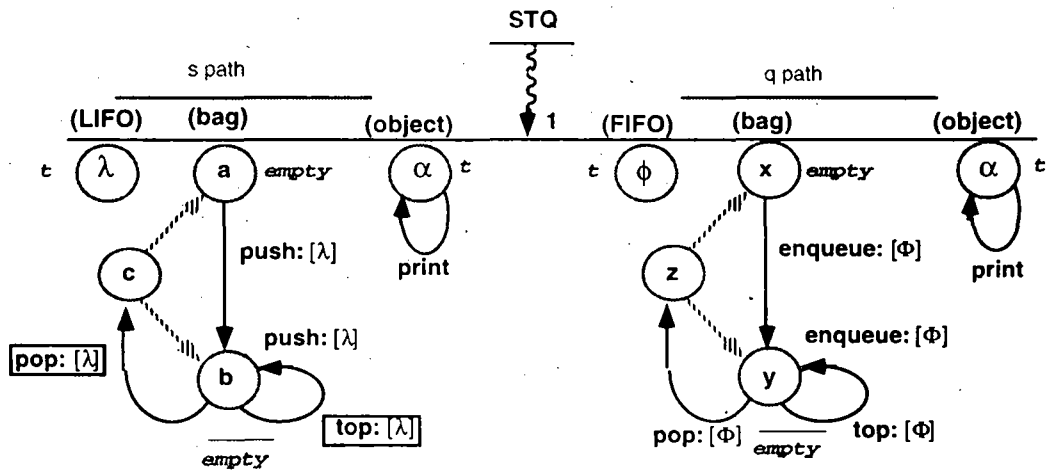


Figure 114.

b.4) Implementing the *print* transition

Let's suppose we want to *print* a *STQ* instance in a given state. Under the duplication hypothesis, we consider the following dimension diagram. The *object* dimension, the only one involved, is represented by a plain line ; the other dimensions, required but not involved, are shown as dashed lines.

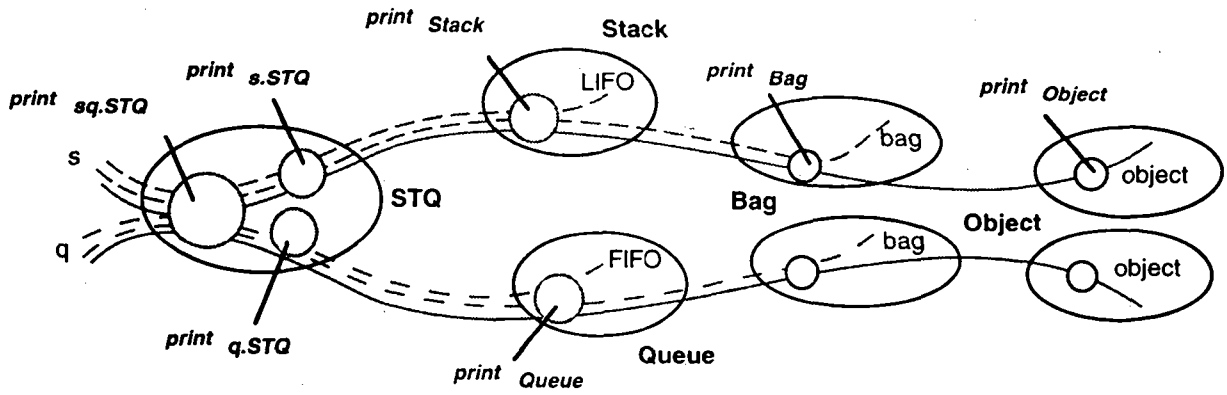


Figure 115.

The figure gets simplified if we use the *stack* and *queue* (resp. noted *s* and *q*) "dimensions" : as explained above, the meaning of *stack* and *queue* is dynamically restricted to the actual dimensions supported by each class .

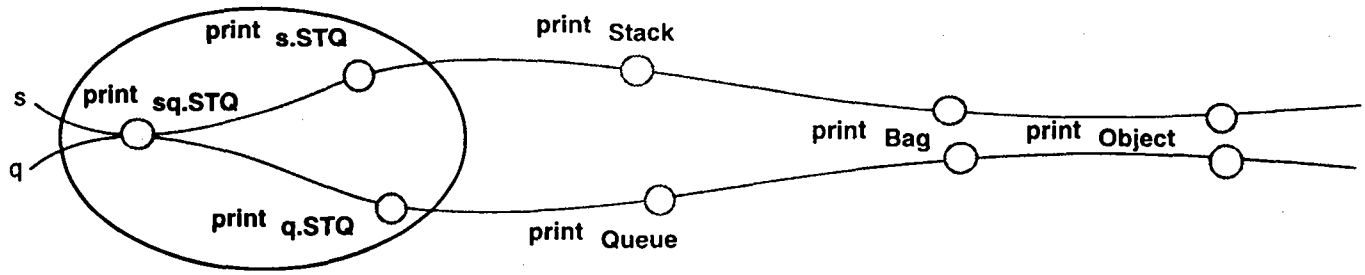


Figure 116.

a) The search starts locally in *STQ*. The "p-graph local search and combination" rule is applied.

a.1) First, *STQ* is looked for a method that will take into account both the *stack* and *queue* dimensions of the instance. If the method $print_{sq.STQ}$ exists, then it is used. The form of the *print* method is thus $print_{sq.STQ}$. (Arguments are not shown here).

a.2) If $print_{sq.STQ}$ is not found, a method is then looked in *STQ* for each of its two dimensions. If $print_{s.STQ}$ and $print_{q.STQ}$ exist, they get combined. The form of the *print* method in *STQ* is then ***(progn*** $print_{sSTQ}$ $print_{qSTQ}$ ***)***.

b) Let's suppose now that both previous steps have failed : the $print_{sq.STQ}$ method is absent as well as at least one of the $print_{s.STQ}$ and $print_{q.STQ}$. What is the form of the combined result if the adequate method(s) is (are) inherited from the *Stack* and/or *Queue* superclasses ?

b.1) Let's suppose that $print_{s.STQ}$ (resp. $print_{q.STQ}$) is absent. The search along the *stack* (resp. *queue*) dimension continues in the involved superclasses of *STQ*, here *Stack* (resp. *Queue*). If $print_{Stack}$ (resp. $print_{Queue}$) is found in this class, it is used in place of the absent *STQ* method for producing the combined method. This inherited method is applied only to the *stack* (resp. *queue*) dimension of the instance, hence the form of the combined method : ***(progn*** $print_{s.Stack}$ $print_{q.STQ}$) (resp. ***(progn*** $print_{s.STQ}$ $print_{q.Queue}$)).

b.2) If both methods of *STQ* are absent and if both $print_{Stack}$ and $print_{Queue}$ are found, then the form of the combined method becomes ***(progn*** $print_{sStack}$ $print_{qQueue}$).

c) Let's suppose that all previous steps have failed in providing a combined method : the $print_{sqSTQ}$ method is absent ; in addition, a method has not been found for the *stack* dimension (both $print_{S.STQ}$ and $print_{Stack}$ are absent) and/or a for the *queue* dimension (both $print_{q.STQ}$ and $print_{Queue}$ are absent). What is the form of the combined result in these cases if an adequate method can be inherited from the *Bag* ancestor class ?

c.1) Let's first suppose that only the search along the *stack* dimension has failed. This search continues in the *Bag* class. If $print_{Bag}$ is found in this class, it is used in place of $print_{S.STQ}$ in *(**progn** $print_{S.STQ}$ $print_q$)* where $print_q$ means either $print_{q.STQ}$ or $print_{q.Queue}$. Hence the combined method *(**progn** $print_{S.Bag}$ $print_q$)*. Conversely, if only the search along the *queue* dimension has failed, the existence of an adequate method in *Bag* yields the following form for a combined method *(**progn** $print_S$ $print_{q.Bag}$)* where $print_S$ is either $print_{sSTQ}$ or $print_{sStack}$.

c.2) Let's suppose now that searches along the *stack* and *queue* dimensions have both failed. If an adequate $print_{Bag}$ method is found in *Bag* , then the form of the *print* method becomes *(**progn** $print_{S.Bag}$ $print_{q.Bag}$)*.

d) Let's finally suppose that all previous steps have failed. If an adequate $print_{Object}$ method is found in *Object* , then the form of the *print* method is *(**progn** $print_{S.Object}$ $print_{q.Object}$)*.

To summarize, the form of the *print* method in *STQ* is *(**progn** $print_S$ $print_q$)* where $print_S$ (resp. $print_q$) is the first method found in the *s* (resp. *q*) list (see above figure). The $print_S$ (resp. $print_q$) method is applied to the *stack* (resp. *queue*) "dimension" of the considered instance. In some cases, $print_S$ and $print_q$ correspond in fact to a same method *print** : this is not to say that the *STQ* instance is printed twice : the form says that *print** is used twice, once for the *stack* dimension and once for the *queue* dimension.

b.5) Computing the memory representation

Concerning memory representations, *Object* is not supposed to provide one (*Object* is an abstract class) ; *Bag* provides one, say a list of items termed *elements*. For the sake of the example, we suppose that this implementation is not refined in subclasses.

Under the duplication hypothesis, we thus consider the following dimension diagram. Abstract dimensions are shown as dashed lines.

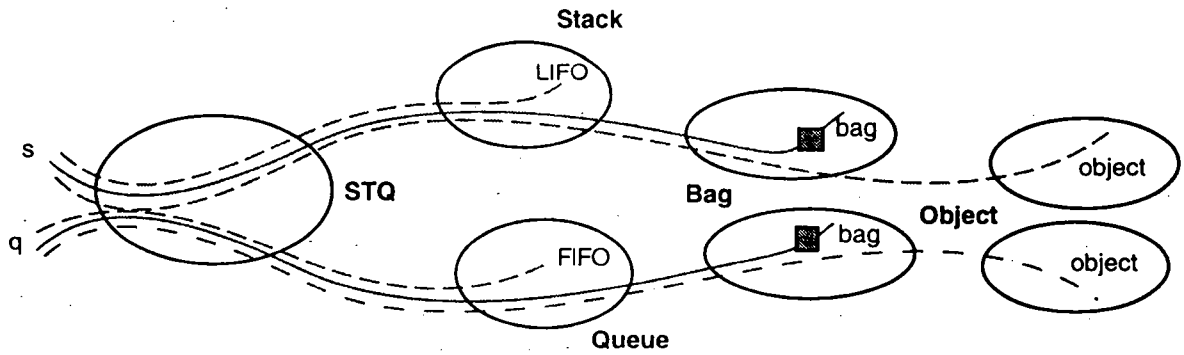


Figure 117.

Since abstract dimensions are non involved when considering memory representations, the "p-graph local search and combination" extended to class inheritance thus concatenates the two *elements* cells.

c) Hypothesis 2 : unification (default)

Let's now consider that a dimension inherited along two or more paths is not duplicated two or more times, but exists only once. Under this hypothesis, only one occurrence of the *bag* or *object* dimensions is now to be considered in *STQ*.

c.1) interpretation in the state space

c.1.1) impact on the number of axes

The unicity of a dimension like *bag* or *object* is easy to interpret in the state space : it simply means the corresponding axis exists but once. Now, the state space has four dimensions (axes) : *LIFO*, *FIFO*, *Bag*, *Object*. Next figure is the analog of figure113 : a given state of an *STQ* instance (say stq_n) is projected onto the four axes of the state space. The order of the axes is important in case masking is used. Hence, the question : what is the order of the dimensions ?

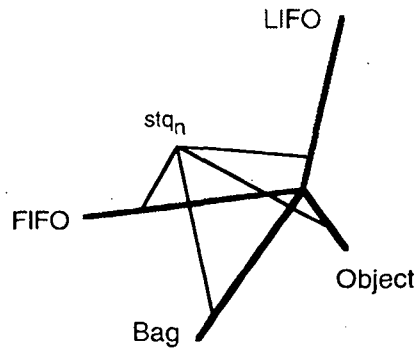


Figure 118.

c.1.2) impact on the bag *and* object *dimension paths*

A subsidiary question is : what classes impact the four dimensions ? As far as the *LIFO* and *FIFO* dimension paths are concerned, there is no difference compared to the previous hypothesis : they both traverse *STQ*, and each one respectively traverses either *Stack* or *Queue*. The *bag* dimension path certainly traverses *Bag*, *Stack*, *Queue* and *STQ* as before. The *object* path traverses the same classes plus the *Object* class. Hence the question : what is the relative order of *Stack* and *Queue* in these two paths ? Next figure shows the four dimensions (axes) together with the classes that impact them (virtual superclasses are included for completeness ; virtual subclasses are named like actual classes).

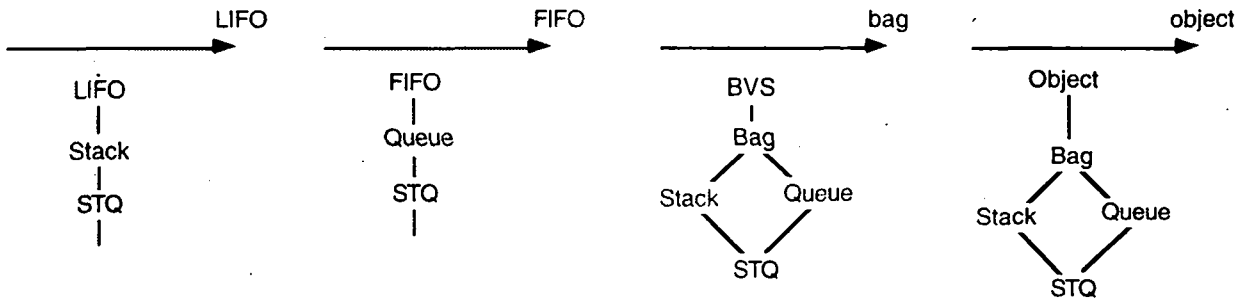


Figure 119.

c.2) determining the orders

We have to determine two orders : (1) the order of dimensions ; (2) the order of classes in the *bag* and *object* dimensions. Shortly , we will see these orders may be obtained in a simple and systematic way once computed the linearization of the hierarchy of *STQ*. For the moment, we start from scratch using but the knowledge accumulated so far.

c.2.1) preliminary

Next figure shows the tree of inheritance paths (termed **TIP**, for short) obtained in a bottom-up, left to right traversal. This traversal has two merits : (1) starting from the root class (here, *STQ*), it quickly collects all ancestor classes out of the complete collection of classes a system may support or an application may need ; (2) the tree organization of all these classes respect (1) the relative ordering of a subclass vs. its superclass, and (2) the ordering of classes stated by the superclass lists. The first type of relation, be it **SUB**, is represented by a subclass beneath its superclass ; the second type of relation, be it **SUPER**, is represented by an horizontal arrow (labelled α, β or γ in the figure).

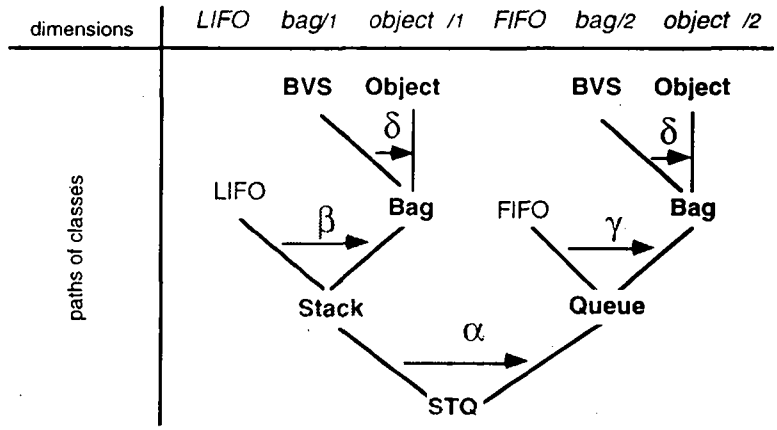


Figure 120.

Note, by the way, that this TIP is the same tree than the one used under the duplication hypothesis (figure 112). Next two tables show in a different manner these same paths of classes (in the order obtained from the TIP). Hence, these two tables express the ordering of dimensions and the ordering of classes per dimension that exist under the duplication hypothesis.

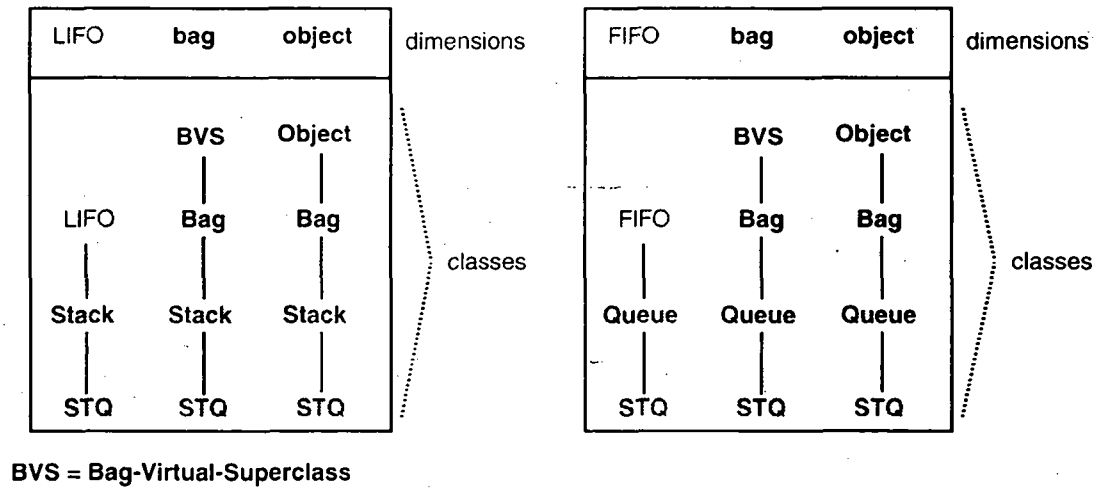


Figure 121.

c.2.2) order of dimensions

Figure 120 lists the dimensions in order : *LIFO* is before *bag* and *object* (due to β) ; *FIFO* too (due to γ) ; *LIFO* is before *FIFO* (due to α) and *bag* is before *object* (due to δ) :
– α expresses that the class *Stack* is before the *Queue* class since both are listed that way in the superclass list of *STQ* ;
– β (resp. λ) expresses that the virtual superclass *LIFO* (*FIFO*) is before the *Bag* class ; in other terms, in *Stack* (resp. *Queue*), *LIFO* (resp. *FIFO*) is a local dimension whereas *bag* and *queue* are inherited dimensions ;
– δ expresses that the virtual superclass *BVS* is before the *Object* class ; in other terms, in *Bag* , *bag* is a local dimension whereas *object* is an inherited one.

Next figure shows the corresponding **precedence graph**. Recursively taking the node to which no arrow points determines the order of dimensions : *LIFO*, *FIFO*, *bag*; *object*.

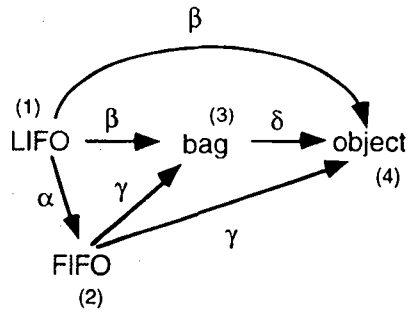


Figure 122.

Here is a systematic way to obtain the same result considering columns obtain by sucessive top down, left to right descents. If a dimension (column) is initially placed before (resp. after) the last occurrence of a dimension to be merged, this dimension should also be placed before (resp. after) the aggregated dimension in the new ordering. In addition, dimensions to be placed before (resp. after) a same aggregated dimension should be in the same relative order as initially. When several dimensions are to be merged, this process is to be done recursively.

Next figure shows this for the *STQ* example. (Note that "dim. to be merged" is to be understood as "occurrences of a same dimension that are to be merged"). Two occurrences of the *bag* dimension exist (fig. 120 or 121). These two occurrences are to be regrouped. (Same remark for the *object* dimension.) The *LIFO* dimension is placed before the first occurrence of *bag* (resp. *object*) : in the new ordering, it should be placed before the aggregated *bag* (resp. *object*) dimension (this preserves the β relation). Similarly, in the new ordering, the *FIFO* dimension shoud be placed before the aggregated *bag* (resp. *object*) dimension (this preserves the γ relation). The relative ordering of *FIFO* and *LIFO* is kept (this preserves the α relation). Two occurrences of the *object* dimension are counted among the dimensions that were placed after the *bag* occurrences : the merging process is repeated at their level, although in a very simplified manner (the "before" dimensions have already all been taken into account ; and no "after" dimensions exist).

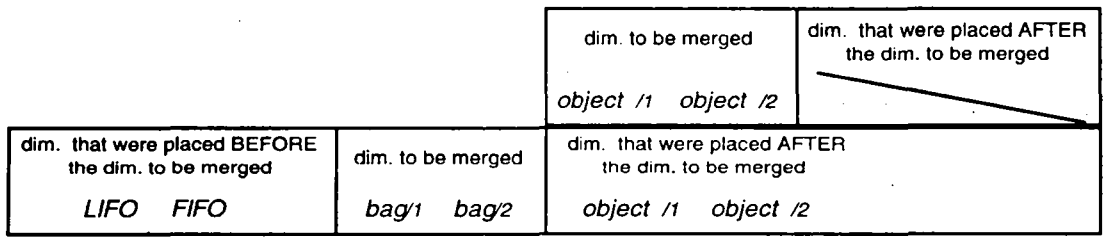


Figure 123.

c.2.3) order of classes for the bag *and* object *dimensions*

To determine the relative order between the *Stack* and *Queue* classes in the *Bag* and *Object* paths, we consider the order of these two classes in the list of superclasses of *STQ* : *Stack* is first and *Queue* is second (relation α). This order is taken advantage of for masking : thus, to not alter this possibility, *Stack* should be listed before *Queue* when going up in the *bag* and *object* dimension paths.

Let's put this in a more algorithmic manner. First, we extract —in order— all the columns corresponding to a same dimension. Next figure shows this for the *object* dimension. (We could have considered the *bag* one instead).

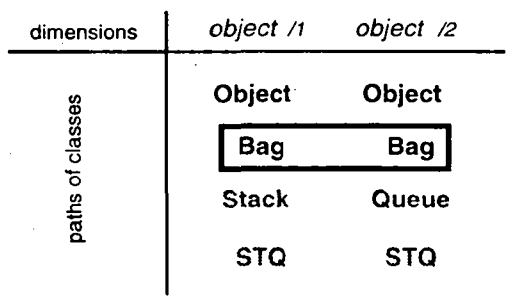


Figure 124.

In this table, *Bag* is encircled as a reminiscence that it is a diverging node : it cannot be listed before it superclass. The goal is here to list each class once and only once, while fulfulling the masking capabilities : these are due to the SUB and SUPER relations. As explained above, the tables from which this figure is extracted already obey these ordering ; since the two extracted columns are listed in the same order as in the original tables, the correct relationships hold. What we have to do is to take a superclass after all its subclasses. In addition, when a class has been listed, its other occurrences are eliminated. (This simple algorithm is termed **LIN** for future references.)

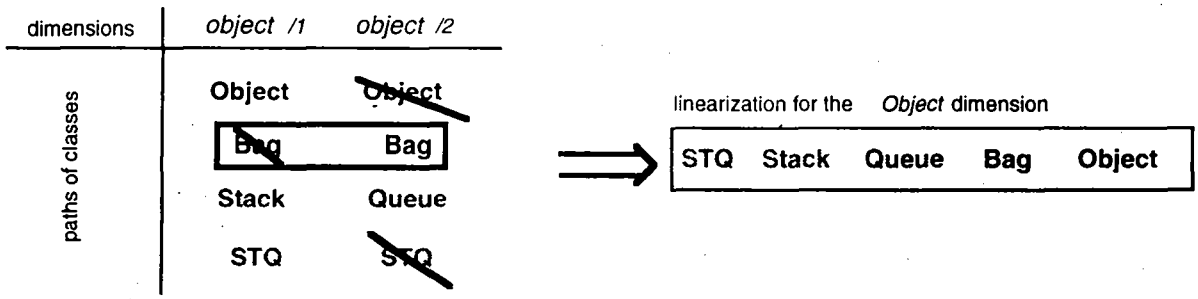


Figure 125.

The algorithm considers the classes bottom up, from left to right. First, *STQ* is selected : its right occurrence is eliminated. Then, going up one level, *Stack* is taken. Going up one more level is not possible since *Bag* requires all its subclasses to be listed before. The sole possibility is to move to the next column (on the right) : *Queue* is taken. Then, *Bag* is taken (one occurrence) and *Object* (same remark). Hence, the result : *STQ, Stack, Queue, Bag, Object.*

c.2.4) summary of the results

Next figure summarizes the results concerning the order of the four dimensions and the order of the classes for the *bag* and *object* dimension (the virtual superclasses have been omitted). Shortly, both results will be obtained at once, in a systematic manner.

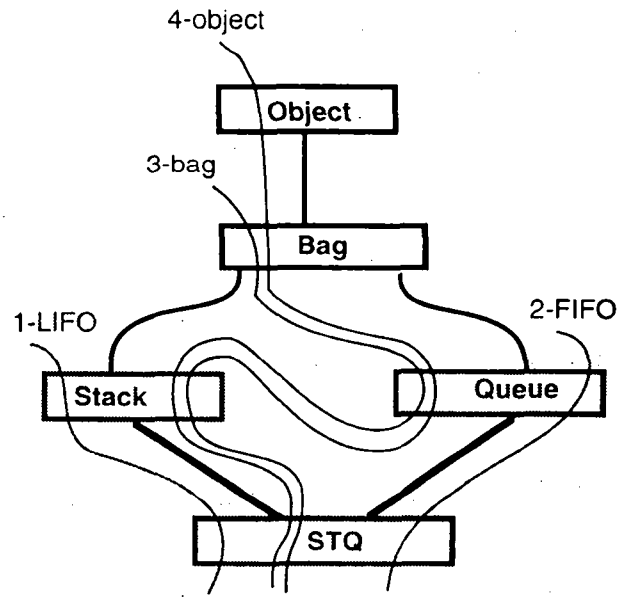


Figure 126.

c.3) impact on the p-graph

Next figure shows the resulting *STQ* p-graph. Dimensions are listed in order. The subgraph for the *bag* dimension directly results from collecting the renamed transitions along the *bag* path. Note that *pop[ϕ]* and *top[ϕ]* are systematically masked by *pop[λ]* and *top[λ]* : they can be removed.

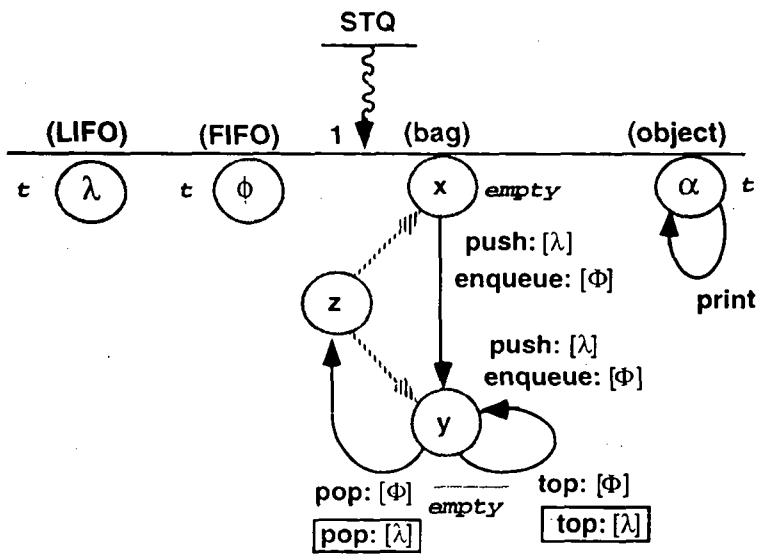


Figure 127.

This p-graph can be augmented with methods and memory representations.

c.4) Computing the memory representation

For the sake of simplicity, we still suppose that the implementation provided by *Bag* (*elements* cell) is not refined in subclasses (the *LIFO* and *FIFO* properties are also abstract). Given the unification hypothesis, this implementation is the only one inherited in *STQ* when following the *bag* path.

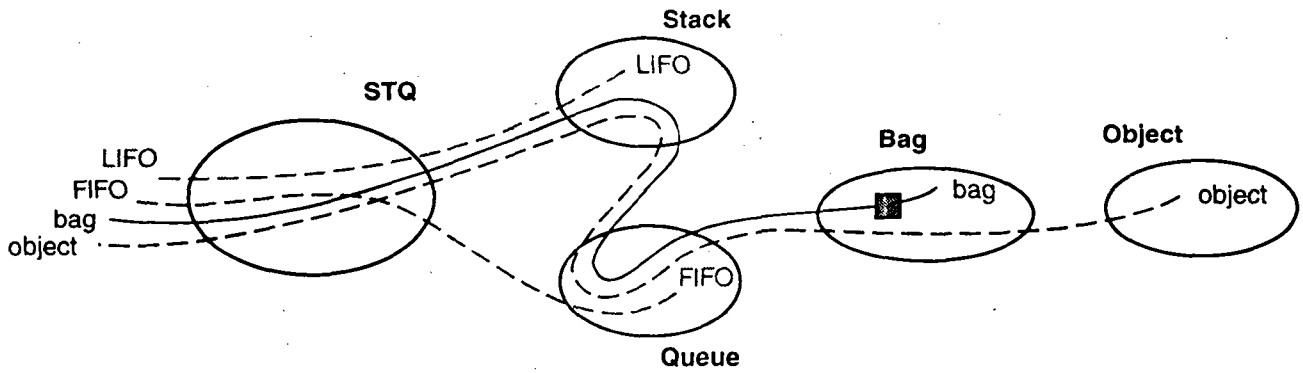


Figure 128.

c.5) Implementing the *print* transition

The inheritance algorithm retains the first method (most specialized one) found going up along the *object* path. Supposing the existence of the same methods as above, the next figure shows how they are ordered for a *STQ* instance in a given state. As mentionned before, the *STQ* hierarchy is not regular vs. the *print* method (cf. §7.3.2.b.2.1.δ).

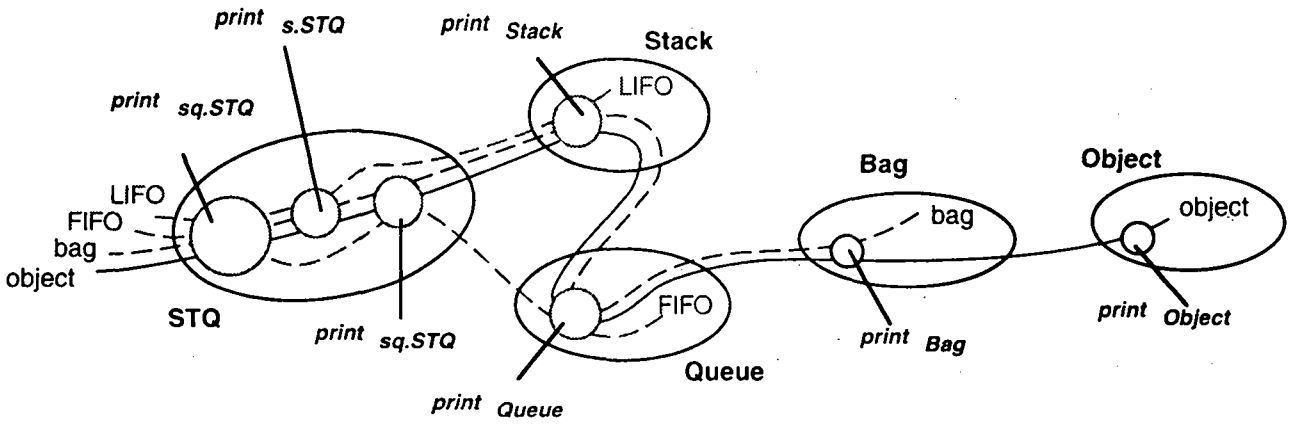


Figure 129.

7.3.4 Inferring the necessary orderings from the linearization of the class hierarchy

In the above two examples, we had to order :
(1) the dimensions, i.e. the axes of the state space ;
(2) the classes impacting each dimension.

Under the duplication hypothesis, the structure we get is a tree and a bottom-up, left to right traversal provides both results.

Consequently, we only consider here the other hypothesis (unification), i.e. the case where a dimension is considered to be unique be it inherited along several class paths. In this case, as already mentioned, a linearization algorithm is runned. Be *C0* the given root class. H_{C0} is the complete class hierarchy rooted in *C0*, and *LC0* the linearization of H_{C0}. Unless specified otherwise, the expression 'the linearization" refers to L_{C0}.

Solving the first problem (i.e. ordering the dimensions) once *LC0* is known is easy. Solving the second one (inferring from *LC0* the order of all classes that impact a given dimension) requires a strong hypothesis on the linearization algorithm.

a.1) Rule 1 : virtual subclasses may be added a posteriori

Considering the previous unification example, one may notice that the virtual superclass corresponding to the addition of a new dimension *d* at the level of a class *D* can easily be taken into account in a second step. Such a virtual class (ex. : *LIFO*, *FIFO*, *BVS*) should appear immediately after its subclass (resp. *Stack*, *Queue*, *Bag*), and before the actual superclasses (resp. *Bag*, *Bag*, *Object*).

a.2) Rule 2 : the order of dimensions may be inferred from the linearization

Dimensions (ex. : *LIFO*, *FIFO*, *bag*,...) are ordered according to their respective virtual superclass (ex. : *LIFO*, *FIFO*, *BVS*, ...). Given the preceding rule, the order of these virtual superclasses is directly obtained from their respective immediate subclass (ex. : *Stack*, *Queue*, *Bag*, ...). Each virtual superclass (ex. : *LIFO*) precedes the actual superclasses (ex. : *Bag*) : thus, each dimension (ex. : *LIFO*), precedes the other dimensions (ex. : *bag*, *object*) that are inherited in the subclass in question (ex. : *Stack*).

If we consider the previous *STQ* example, *LC0* is the list *(STQ Stack Queue Bag Object)*. Thus, the order of dimensions is (*LIFO FIFO bag object*). (There was no dimension introduced by *STQ*.)

We implicitely considered so far that a class was introducing at most one dimension. As a matter of fact, a class may well introduce more dimensions : as already said (subsection 7.1.1), the dimensions are primarily sorted by the order of superclasses in the list of superclasses, and secondarily by the order of dimensions in superclasses exhibiting several dimensions. There is thus no difficulty to handle any number of dimensions.

a.3) Rule 3 : the order of classes for any dimension may be inferred from a linearization if congruent

a.3.1) sub-hierarchy

Let's note $L_{C0}(d)$ the linearization of the classes corresponding to the d dimension. Given rule 1, we can omit the virtual superclass itself in a first step and re-introduce it afterwards. Doing this makes any hierarchy considered for a given dimension d a **sub-hierarchy** $h_{C0}(D)$ of the whole hierarchy. This one contains but the classes found on all paths leading downwards from D to C0. Once obtained $LC0(h_{C0}(D))$ —noted $LC0(D)$ for short, $L_{C0}(d)$ is immediate.

a.3.2) congruency

For any D class (introducing a dimension d), we want the restriction of L_{C0} to the classes belonging to $h_{C0}(D)$ to be equal to $LC0(h_{C0}(D))$. We term this property **congruency**. In general, a linearization algorithm (ex. : the CLOS one) does not respect this property for any hierarchy it is able to process.

Let's suppose the linearization algorithm for COP satisfies this property.

$$\forall D \in H_{C0}, \text{proj}\ (L_{C0}, \text{classes}\ (h_{C0}(D)) = LC0(h_{C0}(D)).$$

The process is then the following : (1) given a dimension, find all classes of its sub-hierarchy ; (2) then take the linearization L_{C0} and remove all classes that do not belong to the considered sub-hierarchy ; (3) append the virtual superclass —if one exists— at the end of the remaining classes (note that no virtual superclass is associated to the *object* dimension).

Note that if an algorithm is congruent, then the following property also holds : for any two classes of the hierarchy H_{C0} having a common part below a given class, the algorithm delivers the same order for the common part (for the linearization of the two associated sub-hierarchies). We term this property **harmony**.

a.3.3) example

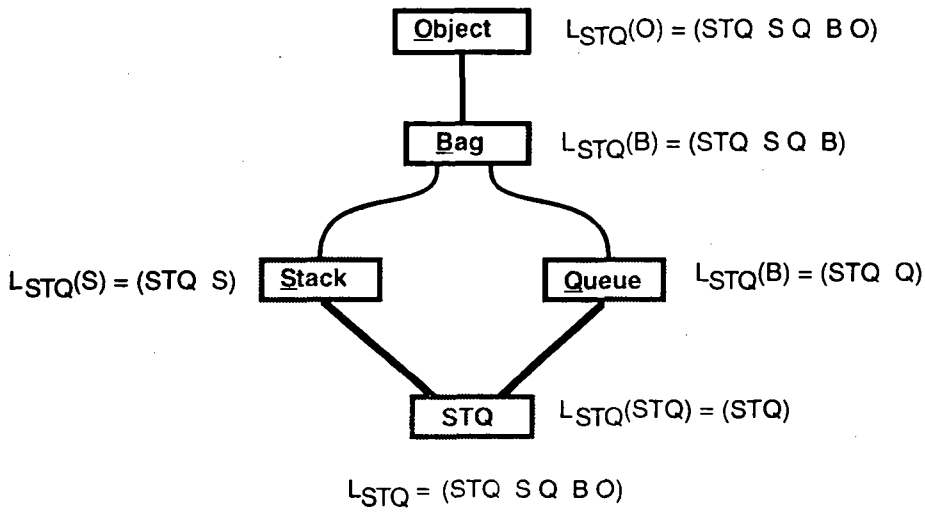


Figure 130.

If we consider the previous *STQ* example, *C0* is *STQ* ; H_{C0} is H_{STQ}, i.e. the list *(STQ Stack Queue Bag Object)*. Previous figure indicates the (direct) linearization of each sub-hierarchy having *STQ* as root node. For example, the sub-hierarchy $h_{STQ}(Stack)$ is made of classes *Stack* and *STQ* (computed downwards) ; its linearization is *(STQ Stack)*. Now, let's compute the corresponding restrictions of L_{STQ}. For the *LIFO* dimension, the classes to be considered are *LIFO* (virtual superclass), *Stack* and *STQ*. The restriction of L_{STQ} to the classes of $h_{STQ}(Stack)$ is the ordered list *(STQ Stack)*. We get the correct result since it is equal to $L_{STQ}(Stack)$. To obtain *L(stack)*, we simply add the virtual superclass : *(STQ Stack LIFO)*. It is easy to yield and check all other cases (i.e. dimensions). See next figure.

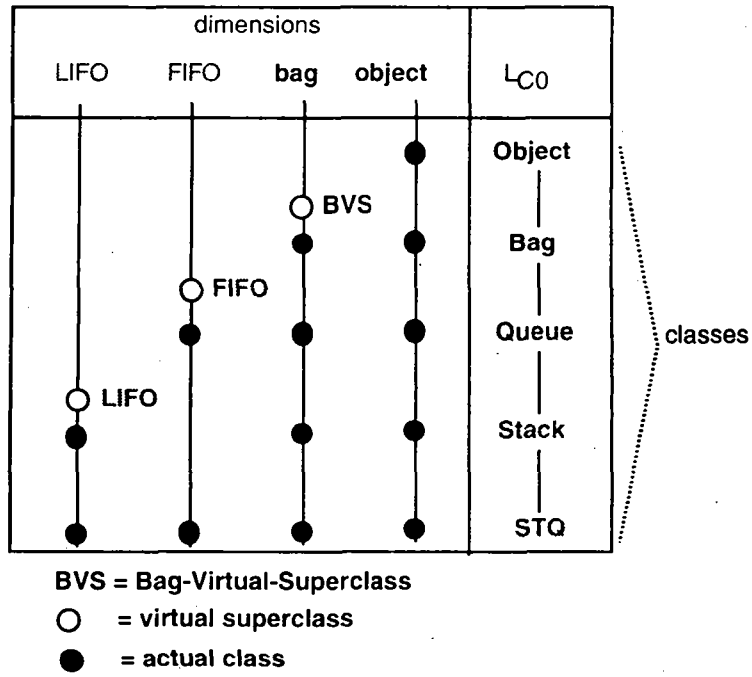


Figure 131.

Note. In the *STQ* example, congruency is obtained whatever the linearization algorithm. (We implicitely consider here that any linearization algorithm does respect the SUB and SUPER relations.) The SUB relations (between a subclass and a class) and the SUPER relations (between the superclasses of a class) sufficiently constrain the hierarchy (hence, L_{STQ}) and each sub-hierarchy $h_{STQ}(X)$ (hence, $L_{STQ}(h_{STQ}(X))$ where X is any class of the *STQ* hierarchy. Suppressing from H_{STQ} the nodes that do not intervene in a given sub-hierarchy does not perturb these relations. Hence, these two types of relation make, in this example, the order of the classes in the sub-hierarchy linearization the same than the relative order of the classes in the whole hierarchy linearization.

7.3.5 On the choice of a linearization algorithm

a) Criteria

When deciding which properties our linearization algorithm should have, we face the following list :

— **uniformity** : the algorithm does not depend on the semantics of the items (memory representation, micro-method) ;

— **stability** : adding an intermediate class between a class and its (direct) superclass should not perturb the order of classes (the restriction of the new ordering to the previous classes should yield the previous ordering) ;

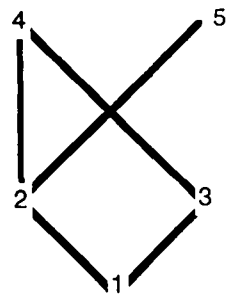
— **monotonicity / incrementality** : once a programmer understands the global behaviours of the superclasses of a class, this programmer can infer the global behaviour of this class simply by composing its incremental behaviour with the global behaviour of its superclasses. In particular, when defining a new class, the programmer is solely required to check the global behaviour of the direct ancestors (superclasses) of this class without having to browse its whole hierarchy. More precisely, if a class inherits an item from an ancestor class different from one of its superclasses, one of these should inherit this item too (otherwise, the result appears relatively surprising). As a consequence, the ordering obtained for any superclass of *C0* should be a sublist of *LC0* . For more details, see [Ducournau et alii, 1994].

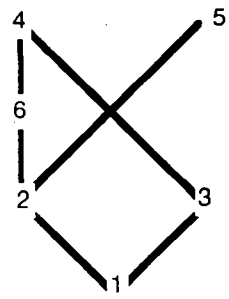
— **congruency** : see above ;

b) Counter-examples

A linearization usually does not fulfil the last three properties for any hierarchy it can process. For exemple, the CLOS one does not.

b.1) stability





$L1_{CLOS}$ = (1 2 3 4 5) $L'1_{CLOS}$ = (1 2 6 5 3 4)

Figure 132.

The ordering of nodes 1 to 5 is perturbated by the insertion of node 6 between nodes 2 and 4. The restriction of L'1 to nodes 1 to 5 does not yield the same order as L1.

Next figure shows the values of L1 and L'1 for other linearization algorithms : LOOPS, the one proposed by [Ducournau et alii, 1994], and finally the one we propose. All are stable.

$L1_{LOOPS}$ = (1 2 5 3 4) $L'1_{LOOPS}$ = (1 2 6 5 3 4)
$L1_{DHHM}$ = (1 2 3 4 5) $L'1_{DHHM}$ = (1 2 6 3 4 5)
$L1_{BOR}$ = (1 2 3 4 5) $L'1_{BOR}$ = (1 2 6 3 4 5)

Figure 133.

b.2) monotonicity

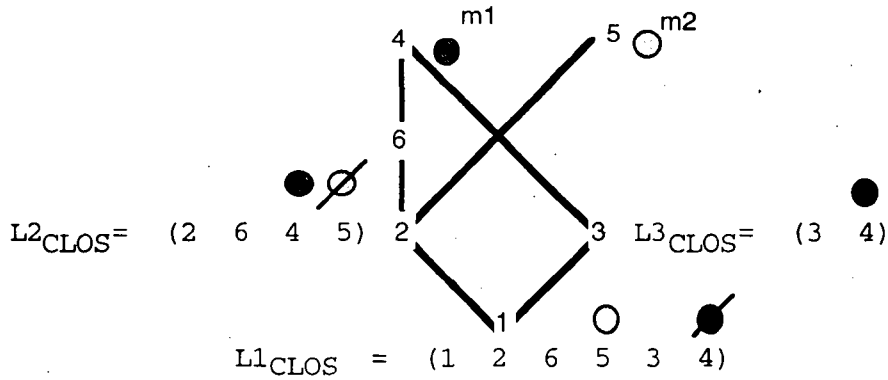


Figure 134.

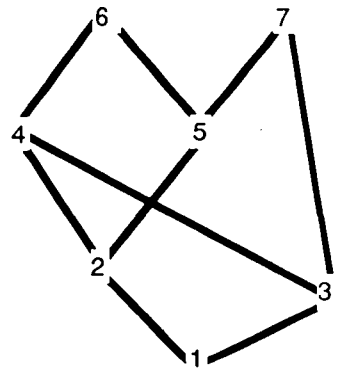
We consider the same inheritance graph as above. We consider methods : method *m1* is attached to node (class) 4 and *m2* to node (class) 5. No other method exists. Method *m1* is inherited in nodes (classes) 2 and 3. Yet, node (class) 1 inherits method *m2*. This is because the ordering in 1 is perturbated when compared to the orderings in 2 and 3.

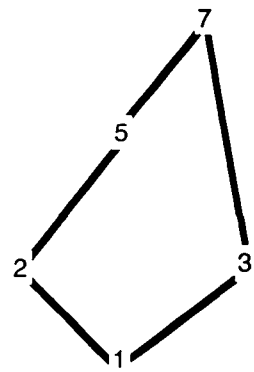
Next figure shows the values of L1, L2 and L3 for the other three linearization algorithms. The LOOPS algorithm is not monotonic in this case ; the others two are.

$L1_{LOOPS}$ = (1 2 6 5 3 4) $L2_{LOOPS}$ = (2 6 4 5) $L3_{LOOPS}$ = (3 4)
$L1_{DHHM}$ = (1 2 6 3 4 5) $L2_{DHHM}$ = (2 6 4 5) $L3_{DHHM}$ = (3 4)
$L1_{BOR}$ = (1 2 6 3 4 5) $L2_{BOR}$ = (2 6 4 5) $L3_{BOR}$ = (3 4)

Figure 135.

b.3) congruency





$L1_{CLOS}$ = (1 2 3 4 5 6 7) $L1_{CLOS}(7)$ = (1 2 5 3 7)

Figure 136.

In this example, the ordering for H1(7) (sub-hierarchy of H1 with summit 7) cannot be obtained by restricting LH1 to the nodes belonging to H1(7) : this restriction gives (1 2 3 57) where the correct value is (1 2 5 3 7).

Next figure shows the values of L1 and L1(8) for the other three linearization algorithms. The LOOPS algorithm is congruent in this case[44] (it is congruent in general : see next subdivision) ; the others two are not.

$L1_{LOOPS}$ = (1 2 5 3 4 6 7) $L1_{LOOPS}(7)$ = (1 2 5 3 7)
$L1_{DHHM}$ = (1 2 3 4 5 6 7) $L1_{DHHM}(7)$ = (1 2 5 3 7)
$L1_{BOR}$ = (1 2 3 4 5 6 7) $L1_{BOR}(7)$ = (1 2 5 3 7)

Figure 137.

c) The LOOPS linearization algorithm is congruent

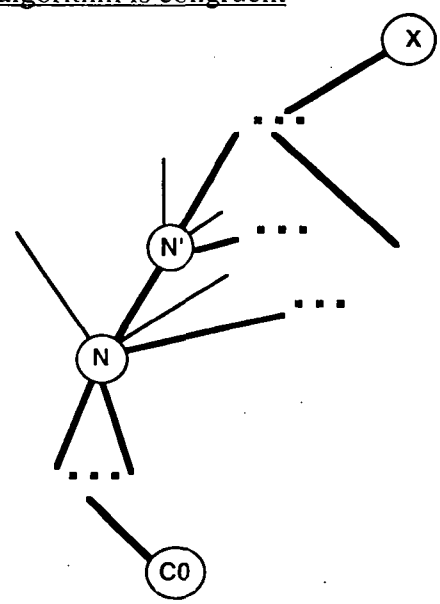


Figure 138.

Let's consider a node *N* on the leftmost path of the sub-hierarchy *HC0(X)* and suppose we are computing *LC0*, the linearization of the entire hierarchy *HC0* rooted in *C0*. When *N* is listed, all its descendants have already been listed too. Be *N'* the first superclass of *N* in *HC0(X)*. All superclasses of *N* may not be part of *HC0(X)*. Thus, branches may exist on the left and on the right of *NN'*.

c.1) Visiting the left branches.

[44] In this case, the LOOPS result is not monotonic : L2 (2 4 5 6 7) is not a sublist of L1.; the CLOS result is monotonic.

Two case depending on whether they exist or not.

— H1 : Let's suppose that no branch exists on the left of *NN'*. Then *N'* is to be visited just after *N* has been listed. The same event would happen if we were computing *HC0(X)*.

— H2 : Let's now suppose branches exist on the left of *NN'*. Having just listed *N*, the LOOPS linearization algorithm visits (attempts to list) the leftmost branch rooted in *N*. None of the visited nodes can perturbate *HC0(X)* :
1) the order of the classes listed so far in *LC0* or *LC0(X)* is left unchanged : once listed, a class is kept ;
2) no visited class on this branch (be it listed or not listed in *LC0*) is to be added to *LC0(X)* : visited classes do not belong to *HC0(X)* (if it were the case, then the branch in question would be part of *HC0(X)* which is contrary to our hypothesis) ;
3) the order of the classes not listed so far in *LC0(X)* is left unchanged too. Visited classes are not subclasses of any node of *HC0(X)*, otherwise they would be part of it. Thus, no unlisted class of *HC0(X)* had its status changed (it has still the same number of subclasses to be listed before itself).
Once this branch has been visited, the linearization algorithm is back to *N* (bactracking).
Thus, from the point of view of *HC0(X)*, all happens as if this leftmost branch was not existing.

By recurrence, all branches on the left of NN' are considered to not exist.
Once all these branches have been visited, the algorithm is again back to *N* (bactracking) and since *N'* exist, it has to be visited.
From the point of view of *HC0(X)*, hypothesis H2 makes no difference with hypothesis H1.

c.2) Branch NN'

N is the first subclass of *N'* (no other subclass of *N'* has already been visited otherwise a path ....*NN''*...would be more on the left than path ...*NN'*... which is the leftmost one by hypothesis). The branch *NN'* is visited under the same conditions than for *HC0(X)* :
— H3 : if *N'* has other subclasses, *N'* cannot be listed in *LC0* until all its other subclasses have been visited : the algorithm bactracks to *N*. If we were computing *LC0(X)*, the same thing will happen ;
— H4 : if *N'* is the unique superclass of *N*, then *N'* is listed immediately in *LC0*. If we were computing *LC0(X)*, *N'* will also be listed immediately. Thus the order of *LC0* and *LC0(X)* will continue to agree if they were agreeing so far.
Once *N'* has been listed, the algorithm recursively continues (our proof is now to be applied to *N'* and so on). Finally, the linearization algorithm bactracks to *N*.

c.3) Right Branches

Other branches may exist on the right of *NN'*. If any, these branches are visited by the LOOPS linearization algorithm when back in *N*.

— H5 : Let's suppose no right branch exists. All superclasses of *N* have been visited (and possibly listed). If *N* is the root node *C0*, we are done. Otherwise, the linearization algorithm backtracks. Our proof too.

— H6 : Let's suppose the next branch to visit is *NN''*... where *N''* belongs to *HC0(X)*. Then *N''* is to be visited now. The same event would happen if we were computing *HC0(X)*.

— H7 : Let's suppose the next branch to visit is *NN''*... where *N''* does not belong to *HC0(X)*. As for branches on the left of *NN'*, this branch does not perturb the order of *LC0(X)*. The reasons are the same (see above hypothesis H2). Once this branch has been visited, the linearization algorithm is back to *N* (bactracking). Thus, from the point of view of *HC0(X)*, all happens as if this right branch was not existing.
By recurrence, all branches on the right of *NN'* that do not belong to *HC0(X)* are considered to not exist. Once all these branches have been visited, the algorithm is again back to *N* (bactracking). If no next branch exists, hypothesis H5 applies as if no right branches were found. If another branch exists, say *NN**, then *N** belongs to *HC0(X)*. From the point of view of *HC0(X)*, hypothesis H6 applies as if no right branches were found.

c.4) Recursively running the proof on HC0

Instead of running the linearization algorithm of LOOPS, we modify it so as to (1) remove, on the way, the left and right branches (vs. the last node belonging to *HC0(X)*) ; and (2) list but the nodes belonging to *HC0(X)*). Here is how works this modified algorithm. First, it lists *C0*. Starting from *C0*, it removes the branches on the left of the leftmost branch rooted in *C0* and belonging to *HC0(X)*. When done, it goes up to the first superclass of *C0* and does the same things. And this recursively, up to the first diverging node in *HC0(X)*. Then, it bactracks like the linearization algorithm of LOOPS. The branches of *HC0* that are on the left of the next leftmost branch of *HC0(X)* are eliminated in turn. ...
Progressing and bactracking like the linearization algorithm of LOOPS (in an accelerated way for the eliminated

branches), the modified algorithm finally lists the summit *X*. Given the points above, the classes listed so far are known to be ordered exactly like they would be in *LC0*. At this point, the pruned hierarchy is *HC0(X)* plus a number of right branches : these are eliminated when bactracking to *C0*. This bactrack does not modify the obtained list of classes.
To summarize, running the linearization algorithm of LOOPS after having eliminated all the left and right branches vs. the nodes of *HC0(X)* yields the same result than the restriction of *LC0* to the nodes of *HC0(X)*. This algorithm is thus congruent.

d) Congruency and monotonicity are antagonistic properties

Let's be more specific : a linearization algorithm which is monotonic and respects the extended precedence order and/or bactracks as less as possible (as done in practical algorithms) is not congruent. For this, we consider the hierarchy shown in figure 136. In addition to the SUB and SUPER relations between nodes, we add the constraint of monotonicity : the ordering obtained at a superclass should be a sublist of the ordering obtained at its subclass(es).

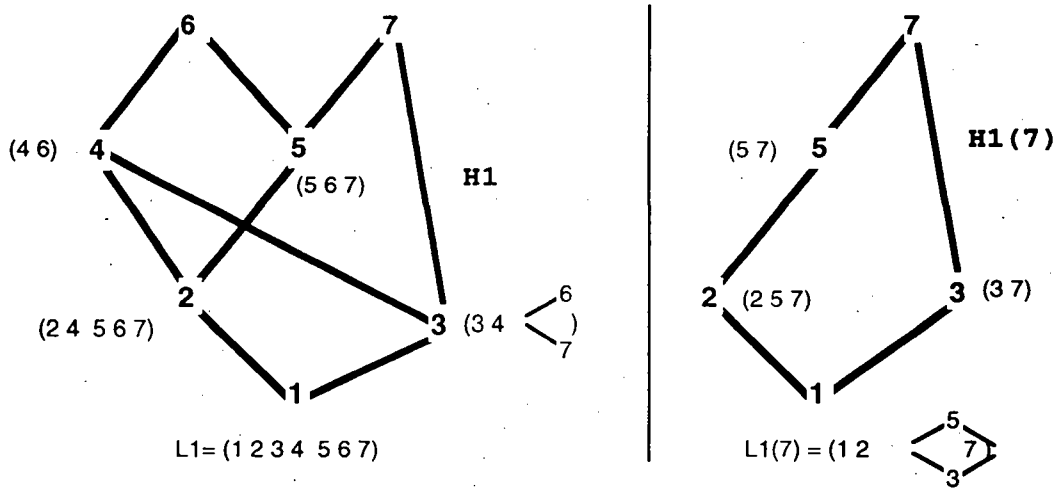


Figure 139.

The figure shows all possible orderings obtained in descending the hierarchy *H1* and its sub-hierarchy *H1(7)* as well. For *H1*, the orderings are total (whatever the monotonic linearization algorithm) in all nodes except node 3. In this node, there are two possibilities (3 4 6) and (3 4 7). Considering *H1(7)*, the orderings are total in all nodes except node 1. In this node, two possibilities exist : (1 2 3 5 7) and (1 2 5 3 7). Instead of listing all the orderings that are valid in each node, the figure shows the precedence diagram these possibilities originate from (these diagrams are constructed top-down and simplified when possible).

L1 and L1(7) do nor conflict if the linearization of H1(7) is (1 2 3 5 7). All the common linearization algorithms do not satisfy this condition, be they monotonic (DHHM) or not (LOOPS, CLOS). The algorithm we propose below has the same behaviour. As a matter of fact, the required ordering is not natural. All common algorithms produce the "natural" ordering (1 2 5 3 7), adopting one of the two following points of view :

— software design point of view : "bactrack as less as possible" so as to list strongly related nodes together, notably when these nodes form a path of ancestors. In our example, the algorithm would thus prefers —once in node 2— to pick up node 5 just above node 2... instead of bactracking to get node 3. This appears quite natural : if the branch 2-5 was the beginning of a tree, it would certainly be listed ; yet, the knowledge that this branch is not part of a tree is beyond node 5 (in node 7 which is diverging). For this reson, we qualify linearization algorithms exhibiting such a behaviour as **blind** algorithms (corresponding property is termed **blindness**) ;

— mathematical point of view : "respect the extended precedence order" : this order is considered as an additional guide for building a sound linearization [Ducournau et alii, 1994]. In our example, nodes 5 and 3 are not directly comparable, but looking at their maximal common subclass -here, node 1- an order can be decided : node 5 is to be listed before node 3. This order appears more natural than the opposite.

Hence, what appears in any case as a rather natural decision actually prevents the ordering of *H1(7)* to be obtained by restricting the ordering *H1* to the nodes of *H1(7)*. The example is valid for any monotonic linearization algorithm. We thus have exhibited a counter-example that invalidates any hope to build a blind monotonic linearization algorithm which can be at the same time systematically congruent (for all hierarchies). (Note that the blindness constraint may be superfluous. If this was demonstrated, the result would be more general : a monotonic algorithm would not be congruent, be it blind or not blind.)

7.3.6 Proposing a linearization algorithm

The algorithm we propose satisfies stability and monotonicity. It is efficient (linear in time en space vs. the number of nodes). It is different from the monotonic algorithm proposed by [Ducournau et alii, 1994] for OOP but provides the same result[45]. Because such a linearization algorithm may be of interest not only for COP but also for OOP, it is described in a separate report [Borron, 1996x]. Its properties are demonstrated in this report. Here, we simply present it briefly.

a) Algorithm

Our algorithm is designed to enable the tracing of the origin of a given node for understandability purpose. (A cognitive difficulty of linearization algorithms like the CLOS one is that their results are difficult to predict.)

The algorithm is simple (two passes) and is based on a same elementary algorithm. It considers paths of classes. Each path has a direction, the direction of the graph traversal when getting paths (top-down or bottom-up) ; each one is examined in reverse direction (resp. bottom-up or top-down) for listing classes in order (listed classes are always on the left when progressing along the path).

a.1) Elementary algorithm

Derived from the LIN algorithm, it takes a list of class paths and successively climbs up each one until a diverging node or the end of the path is found. When a diverging node is found, the algorithm starts climbing up the next path from the bottom. When progressing along a path, the algorithm examines the elements found on his left and lists them if not already listed (it is a pre-order progression).

a.2) Determining paths

Once the diverging nodes are known and their degree as well, the paths are determined by successive bottom up (or top down), left to right climbings (resp. descents) (Progressing that way and noting at each diverging node how subclasses appear to be ordered enables the detection of conflicting hierarchies (ex. : X <Y on one hand, and Y < X on the other hand): in this case, the user is warned and the computation cancelled.)

a.3) Two-pass algorithm

Let's give first a few definitions. If the subgraph flowing out of a given node is connex, it can be linearized : we term it an **independent** branch. If this branch is found on the left of the leftmost branch leading to a diverging node, it is termed a **left independent** branch. If this branch is found in between two branches both leading to a same diverging node, then it is termed an **in-between independent** branch. The remaining independent branches are but **right independent** ones : they systematically are on the right of the rightmost branch leading to a diverging node.

In a preliminary step, the hierarchy is traversed bottom-up, left to right to determine -out of a global hierarchy- what nodes participate to this hierarchy, what are the diverging ones and their degrees and how they are disposed.

a.3.1) pass 1 : left and in-between independent branches

First, the paths are determined by successive top-down, left to right descents. Then, the elementary algorithm is runned the reverse way (i.e. bottom-up), left to right. (Left and in-between independent branches are recursively visited during this path.)

a.3.2) pass 2 : right independent branches

If nodes remained to be listed, they correspond to right independent branches. First, the paths are determined by successive bottom-up, left to right climbings. Then, the elementary algorithm is runned the reverse way (i.e. top-down), left to right. (Right independent branches are recursively visited during this path.)

---

45 [Ducournau et alii, 1995], p.331) states two results about the "extended precedence order" : (1) this order yields a sufficient condition for making any linearization algorithm respecting it monotonic ; (2) this order implies all linearization algorithms respecting it to produce the same result.

- *a.3.3) linearization result*

Be LIN1 and LIN2 the results of the two passes. The linearization of the whole hierarchy is simply LIN1 with LIN2 appended behind.

b) Example

Next figure shows the skeleton of the hierarchy of a *Professional Player* class. This class inherits from *Employee* and *Player*. *Employee* (resp. *Player*) results from the combination of *Active Individual* (noted *Active*) and *Job* (resp. *Game*). *Active* inherits from *Individual*. (*Individual*, *Job* and *Game* have no superclass.)

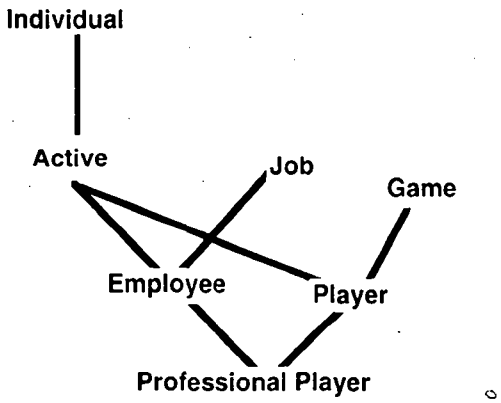


Figure 140.

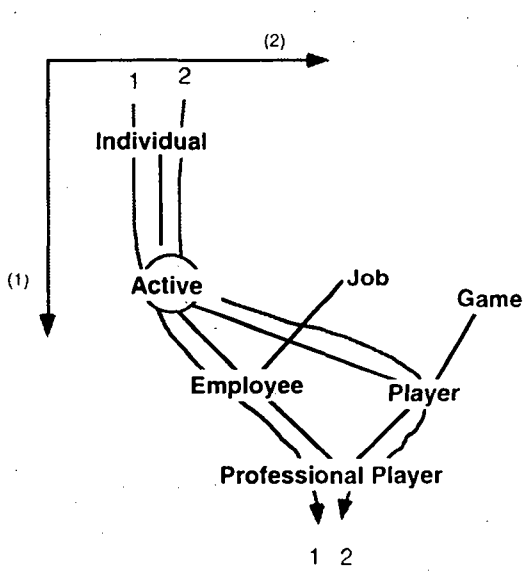
b.1) First pass

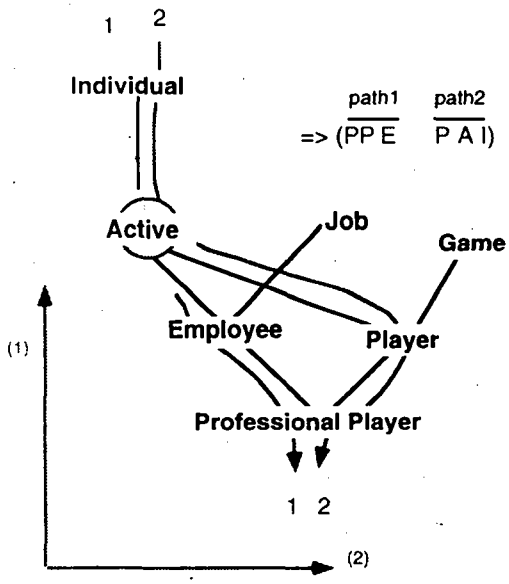
We determine the number of diverging nodes and their respective degree by a top-down, left to right traversal. Hence, the hierarchy disposed in **natural ordering**. (The natural ordering is the one obtained when topologically positioning nodes as they are found for the first time in a bottom-up, left to right traversal (subclass below its superclass(es) ; superclasses of a same subclass disposed from left to right according to the superclass list.)

Here, *Active* is the sole diverging node ; it has a degree 2. *Individual* is the single summit of the hierarchy.

Given this, we determine two paths by a successive top-down, left to right descents. (See next figure.)

The figure afterwards shows how classes are listed. Path1 is climbed up till the diverging node (*Active*). Hence, are listed : *Professional Player* and *Employee*. Then path 2 is climbed up till the summit. Are listed during this progression : *Player*, *Active* and *Individual*.





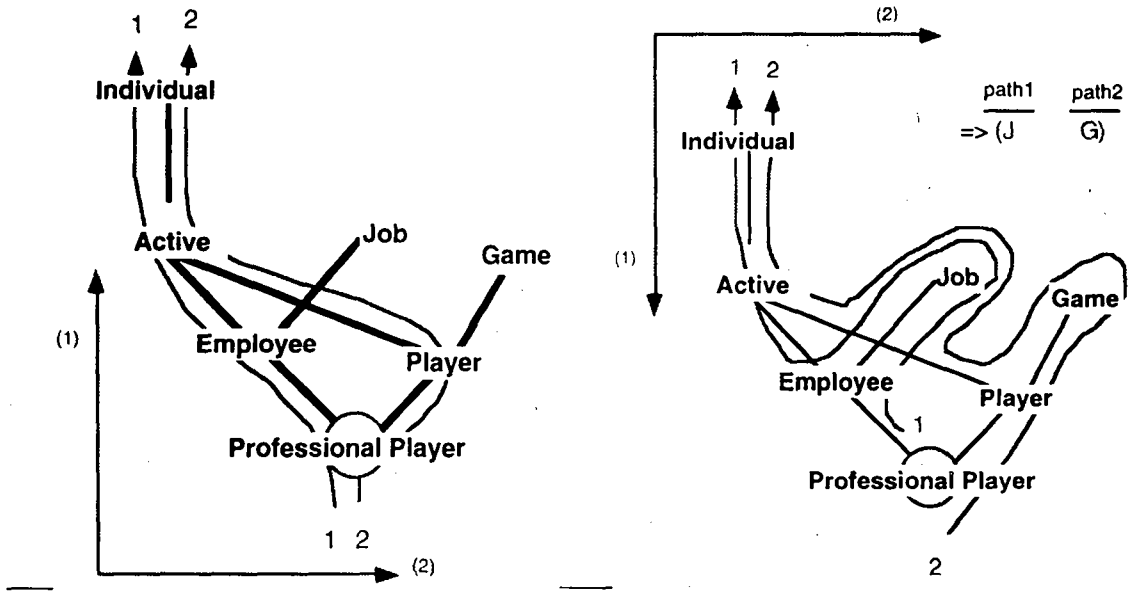
Figures 141 & 142.

b.2) Second pass

Since a few nodes have not been listed, the second pass is executed[46]. The diverging nodes are now the classes that have several superclasses excluding those that correspond to RIGHT branches. Here, a single one exists : *Professional Player.*

Two paths are obtained by a successive bottom-up, left to right climbings. (See next figure.)

The first figure shows how classes are listed. Path1 is descended from *Individual* till the diverging node (*Professional Player*). Hence, is listed : *Job.* Then path 2 is climbed up till the summit. Game is listed during this progression.



Figures 143 & 144.

b.3) Final result

The result of our linearization algorithm is thus (using the first initial of each word) : $Lpp = (PP\ E\ P\ A\ I\ \ J\ G)$.

[46] Because our algorithm lists all nodes in their final order, it can be stopped (in any phase) as soon as all the nodes (minus one) have been found.

PART C : GENERALIZATION

This part explains first how an implementation can be attached to one color graph (local implementation), then how an implementation can be distributed into a hierarchy of color graphs (hierarchical implementation). Hence, respectively, the local inheritance rules for implementations and the class inheritance rules for implementations. This work is done considering first a Smalltalk-style implementation (no combination, systematic masking) and then a CLOS-style one (systematic combination except in case of explicit masking). The solution finally proposed encompasses both styles.

8. SYSTEMATIC CLASS-LEVEL COMBINATION STYLE

So far, masking was systematic ("Smalltalk-style"). We presently come to a different approach ("CLOS-style") where all valid items (ex. : methods) along a same dimension are systematically selected —even when apparently masked— and combined. The nature of this combination is different from our usual combination meant for satisfying several dimensions : in the following examples, the new one is noted with square brackets ([...]) while our usual combination is noted with curly braces ({...}). Obviously enough, there are default combination methods for the new combination (ex. : the standard combination method of CLOS) and ways to prefer other combination methods in place (thanks to MOPs).

To avoid any misinterpretation about "combined" or "combination" or "combination method" in the rest of the text , these terms are used between square brackets in the first case (hence, "[combined]" or "[combination]" or "[combination method]"), and between curly braces in the second one (hence, "{combined}"or "{combination}" or "{combination method}") .

8.1 PRINCIPLE

8.1.1 Description

Here is how the [combination] is done.

Let's first consider the case of an **isolated** dimension (none of its items satisfies also an other dimension). Instead of keeping but one item for this dimension, the whole list of items is first computed : when obtained, all these items are combined in the same order using the [combination method]. The list itself is recursively made by taking the item that would normally be selected in case of systematic masking were the previously selected items not existing.

Let's now consider the case of a group of K **cooperative** dimensions (for any two of these dimensions, exist at least one item that satisfy them both). Such dimensions are processed group by group. The result, for one group, is a [combination] of
— all the items that satisfy all the dimensions, if any ;
— possibly followed by the {combination} of items satisfying the K-1 first dimensions (these are [combined] of course) with items satisfying only the K^{th} (these are also [combined]), possibly followed by the K-1 analog combinations (each one {combines} items satisfying the K-i first dimensions [already combined] with items satisfying only the i^{th} [already combined too] :
— ....
— possibly followed by the {combination} of items satisfying the first dimension (these are [combined]) with items satisfying only the 2^{nd} (these are also [combined]), ... with items satisfying only the K^{th} (these are also [combined]);

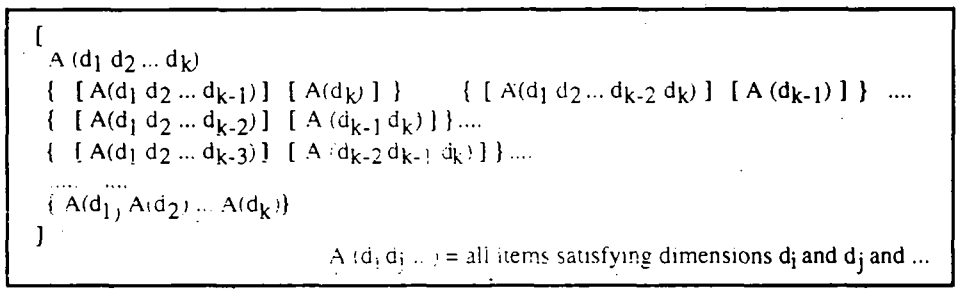


Figure 145

This double-level combination generalizes our most recent algorithm recursively implementing the "p-graph local search and combination" and the "prevalence of combined items" rules. The principles exposed so far are kept : each dimension is satisfied once (vs. a combined method between square brackets now) ; most specialized items are privileged vs. less specialized ones ; the order of dimensions is taken into account.

The proposed computation satisfies two extreme cases :
— the result we get in case of a single dimension is the one that would be obtained in CLOS ;
— the result we get in case of a single item per dimension is the one that would be obtained if masking was systematic.

8.1.2 Impact of regularity

The form of the above formula was established for any hierarchy. If we were to consider but regular hierarchies, then the formula gets simplified since the invocation sequence diagram for a group of cooperative dimensions is a tree (cf. §7.3.2.b : figures 93, 100 and 108). All the $A(d_i\ d_j\ ...)$ are organized as a tree vs the dimensions they satisfy. Compared to the previous formula, we note in particular that : (1) a given $A(d_i\ d_j\ ...)$ may not appear several times in the formula ; (2) two given $A(d_i\ d_j\ ...)$ of same degree may not share (cannot satisfy) a same dimension.

a) Algorithm

The algorithm recursively combines items found along the alleys of the invocation sequence diagram. An **alley** (or **section**) is a set of parallel segments between two diverging points, each segment belonging to the line associated with a dimension. All the items found along a same alley are [combined]. When an alley is splitted into two or more suballeys after point *P*, the results obtained along the suballeys beyond *P* are to be {combined} ; the corresponding result is to be [combined] with items preceding *P* along the alley in question. If an item along an alley is alone, brackets can be omitted.

In the next figure, we can distinguish alleys *Pxyz* (between points *D* and *Dxyz*), *Pxy* (between points *Dxyz* and *Dxy*), *Px* (between points *Dxy* and *Dx*), *Py* (between points *Dxy* and *Dy*), and *Pz* (between points *Dxyz* and *Dz*). Alley *Pxyz* is composed of three segments, one for each dimension *dx*, *dy* and *dz* ; alley *Pxy* is composed of two (for *dx* and *dy*) ; alleys *Px*, *Py* and *Pz*, of one.

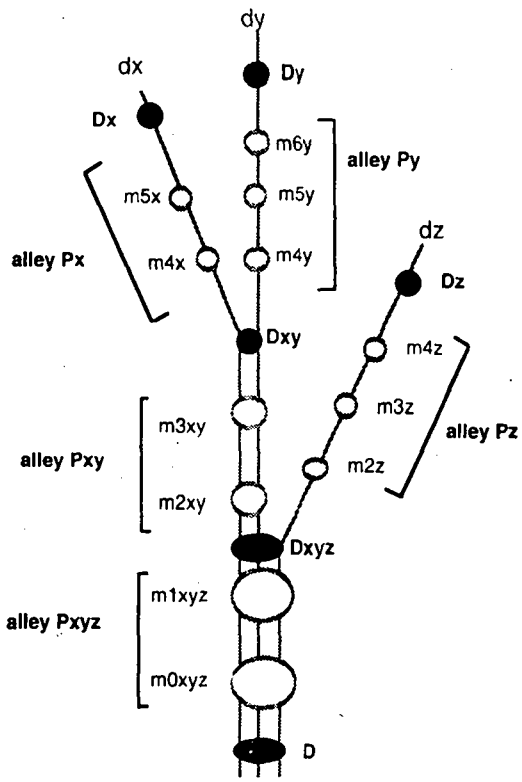
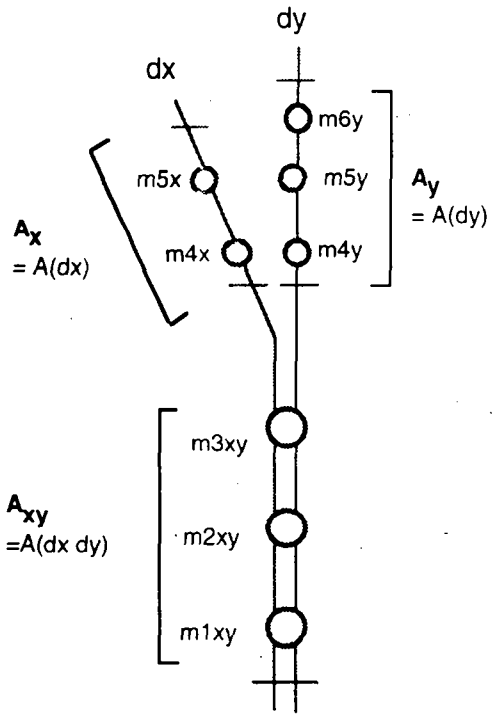
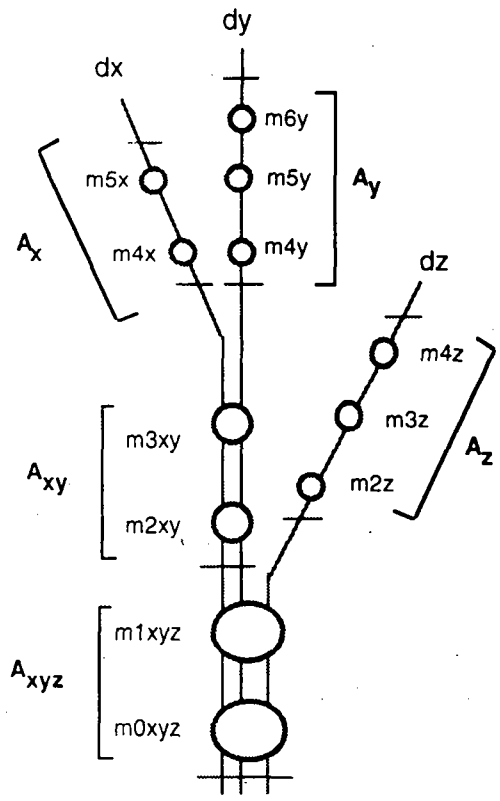


Figure 146.

b) Examples

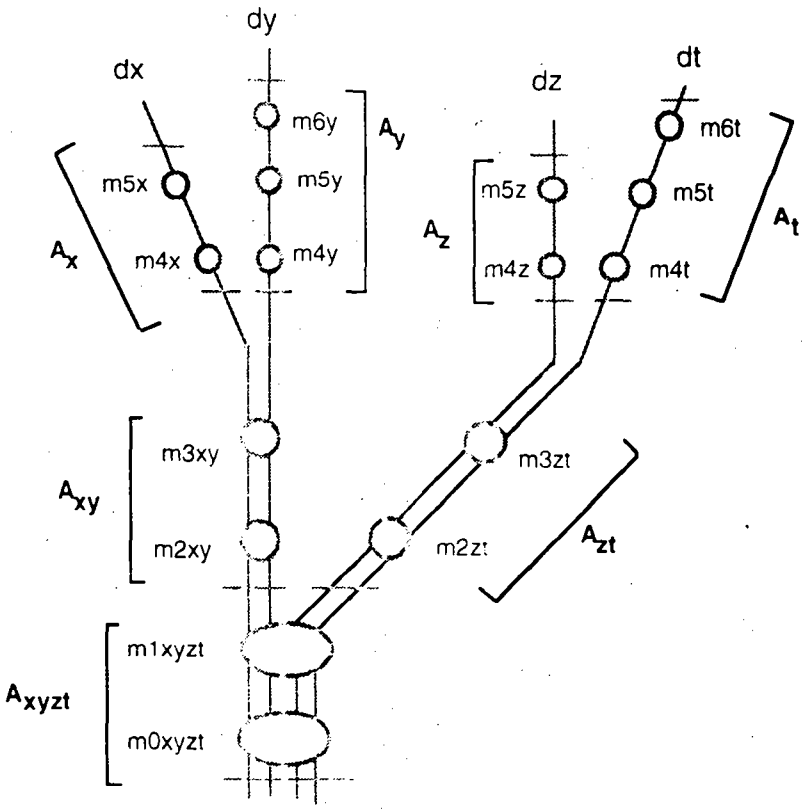
Next figures show a few cases. (Remark : $A(dx\ dy\ ...\ du)$ is noted $A_{xy...u}$.]





$[A_{xy}\ \{[A_x][A_y]\}]$ $[A_{xyz}\ \{[A_{xy}\ \{[A_x][A_y]\}][A_z]\}]$

Figures 147 & 148.



$[A_{xyzt}\ \{[A_{xy}\ \{[A_x][A_y]\}][A_{zt}\ \{[A_z][A_t]\}]\}]$

Figure 149.

The three combinations obtained for the specific figures are shown below.

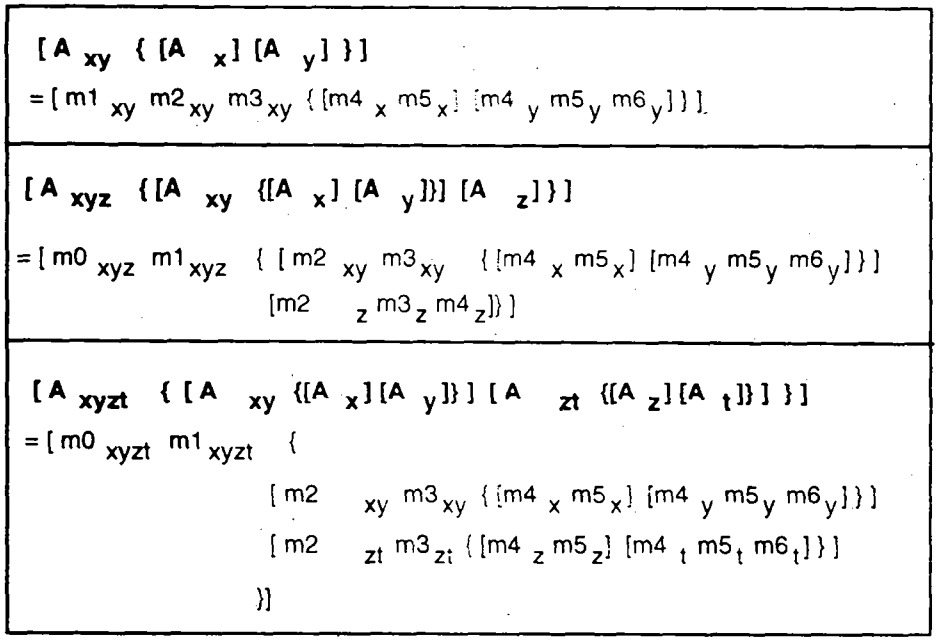


Figure 150

8.1.3 Possible simplification

Were the form in case of regularity considered as too complex for practical programming or were the regularity not obtained, a reduced form may be chosen. This form consists in keeping —for each group of dimensions— but the first *A (di dj ... dn)*, the elements of which are valid for all dimensions (in other words, the sole elements that belong to the trunk[47] of the invocation sequence diagram for the dimensions in question). Of course, this choice requires more work from the user since less automatic combination is done.

8.2 EXAMPLES

In the examples below, the full form is considered. The result is given be the hierarchy regular or not. If not regular, we consider the regular hierarchy that can be derived from it and give also the result for comparison.

8.2.1 Example 1 : tree structure

The table shown in the next figure is established from figure 89 once *m0* has been removed. Dimensions or groups of dimensions are listed in order. For each one. the [combined] item is shown between brackets if it is built out of a list ; and without brackets if single. To obtain the final result. this list of combined items are to be {combined}. (Post-methods, if any, would be listed in reverse order.)

Note that each line in this table is easily obtained from figure 93.

47 i.e. the first alley starting from the root class

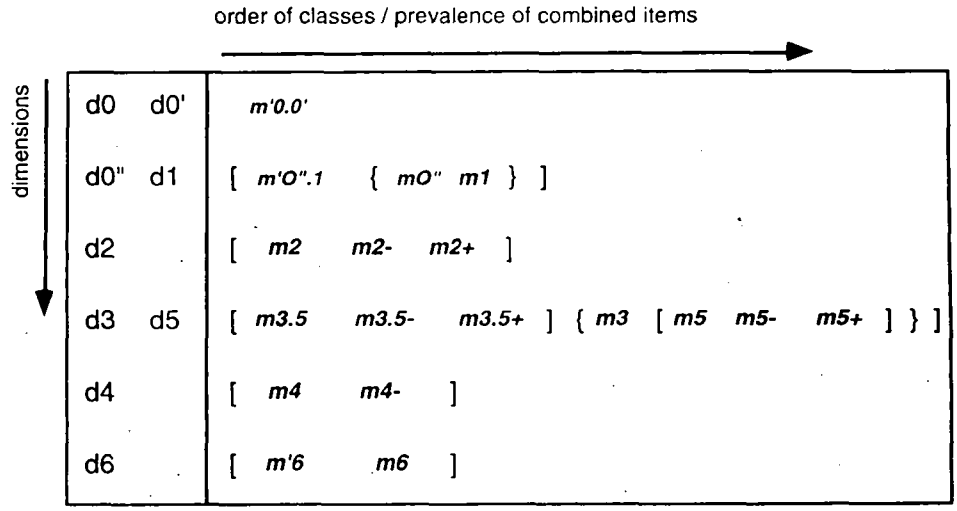


Figure 151.

The result for each set of dimensions may be obtained in an incremental way. In this purpose, *S (di dj... dn)* is used to pick up in the superclasses the part corresponding to the dimensions *di* and *dj* and *...dn*. Figure 153 shows the process for the pair of dimensions *d3-d5* (see corresponding inheritance subgraph in the next figure).

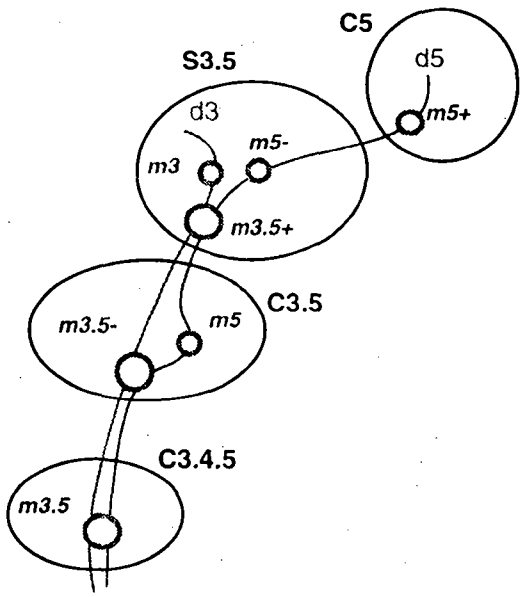


Figure 152.

	formula	value
C_5	[*m5+* ~~S(d5)~~]	*m5+*
$S_{3.5}$	[*m3.5+* S(d3 ~~d5~~) { [*m3* ~~S(d3)~~] [*m5-* S(d5)] }]	[*m3.5+* { *m3* [*m5- m5+*]}]
$C_{3.5}$	[*m3.5-* S(d3 d5) { S(d3) [*m5* S(d5) }]	[*m3.5- m3.5+* { *m3* [*m5* *m5- m5+* }]
$C_{3.4.5}$	[*m3.5* S(d3 d5) { S(d3) S(d5)}]	[*m3:5* *m3.5- m3.5+* { *m3* [*m5* *m5-*]}]

Figure 153.

8.2.2 Example 2 : dag structure, no conflict after linearizations

Here, we take the same examples than in §7.3.2.b.2.1.

a) Non regular hierarchy

Let's consider figure 94. Systematic selection and combination of methods according to the above rules produce the following result. This result is easy to obtain simply by looking at the cited figure. It is a [combination] of the method valid for all three dimensions with : first, the result of the [combination] of the methods specified for both first two dimensions {combined} with the [combination] of methods valid for the sole third dimension ; second, the result of the [combination] of the methods valid for the sole first dimension {combined} with the unique method specified for both last two dimensions ; third, the {combination} of the three [combinations] that can be obtained for each dimension separately.

$$\{\, \mathbf{m0}_{xyz} \; \{\, [\, \underline{\mathbf{m1}_{xy}\; \mathbf{m4}_{xy}}\,]\; [\, \mathbf{m5}_z\; \mathbf{m3}_z\,]\, \}\; \{\, [\, \mathbf{m1}_x\; \mathbf{m4}_x\,]\; \underline{\mathbf{m3}_{yz}}\, \}\; \{\, [\, \mathbf{m1}_x\; \mathbf{m4}_x\,]\; [\, \mathbf{m2}_y\; \mathbf{m3}_y\; \mathbf{m4}_y\,]\; [\, \mathbf{m5}_z\; \mathbf{m3}_z\,]\, \}\, \}$$

Figure 154

The table shown in the next figure is the analog of figure 153. It shows a possible way to incrementally compute the result by asking the superclasses the *S (di dj ...)* values. Note that a class is able to answer only demands for its dimensions (single or combined) : if unable, it asks its own superclasses. Note also that one —and only one— class is responsible for answering a *S (di dj ...)* demand. For example, the demand *S(dy dz)* from *C0* cannot be answered by its direct superclasses : they transfer the demand to their superclasses ; finally, *C3* is able to answer it.

	formula	value
C_0	$[\, \mathbf{m0}_{xyz}\; S(dx\, dy\, dz)\; \{\, \underline{S(dx\, dy)}\; S(dz)\, \}\; \{\, \underline{S(dx\, dz)}\; S(dy)\, \}\; \{\, S(dx)\; \underline{S(dy\, dz)}\, \}\; \{\, S(dx)\; S(dy)\; S(dz)\, \}\,]$	*
C_1	$[\, \mathbf{m1}_{xy}\; S(dx\, dy)\; \{\, [\, \mathbf{m1}_x\; S(dx)\,]\; S(dy)\, \}\,]$	$[\, \underline{\mathbf{m1}_{xy}}\; \underline{\mathbf{m4}_{xy}}\; \{\, [\, \mathbf{m1}_x\; \mathbf{m4}_x\,]\; [\, \mathbf{m2}_y\; \mathbf{m3}_y\; \mathbf{m4}_y\,]\, \}\,]$
C_2	$[\, \mathbf{m2}_y\; S(dy)\,]$	$[\, \mathbf{m2}_y\; \mathbf{m3}_y\; \mathbf{m4}_y\,]$
C_3	$[\, \mathbf{m3}_{yz}\; S(dy\, dz)\; \{\, [\, \mathbf{m3}_y\; S(dy)\,]\; [\, \mathbf{m3}_z\; S(dz)\,]\, \}\,]$	$[\, \underline{\mathbf{m3}_{yz}}\; \{\, [\, \mathbf{m3}_y\; \mathbf{m4}_y\,]\; \mathbf{m3}_z\, \}\,]$
C_4	$[\, \mathbf{m4}_{xy}\; S(dx\, dy)\; \{\, [\, \mathbf{m4}_x\; S(dx)\,]\; [\, \mathbf{m4}_y\; S(dy)\,]\, \}\,]$	$[\, \underline{\mathbf{m4}_{xy}}\; \{\, \mathbf{m4}_x\; \mathbf{m4}_y\, \}\,]$
C_5	$[\, \mathbf{m5}_z\; S(dz)\,]$	$[\, \mathbf{m5}_z\; \mathbf{m3}_z\,]$

(*) *see preceding figure*

Figure 155.

Note that, for each class satisfying a number of dimensions (ex. : *dx* and *dy*) , the "prevalence of combined items" rule consists in placing just after the item of the class satisfying a number of dimensions (ex. : $m1_{xy}$) a *S* expression satisfying the same dimensions (ex. : *S(dx sy)*), the value of this *S* expression being asked to the superclasses. If this item does not exist, the *S* expression should nevertheless be present (ex. : *S(dy)*). If the item exists and is to {combined} with other ones, this item and the *S* expression should be [combined] (ex. : $[m1_x\, S(dx)]$).

b) Regular hierarchy

In this case, we consider figure 98 instead of figure 94. The result is easily obtained from the invocation sequence diagram of figure 100. It corresponds to the case shown in figure 148. In the above formulas, regularity enables simplifications : in C0, only one of the expression *S(dx dy), S(dx dz), S(dy dz)* may be not empty.

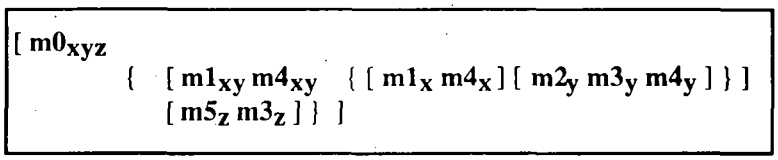


Figure 156.

c) Simplified form

Were the simplified form to be used, the combined method will be reduced to $m0_{xyz}$ in any case.

8.2.3 Example 3 : dag structure, conflicts after linearizations

In this subsection, we draw our examples from subdivision §7.3.2.b.2.2.

a) Non regular hierarchy

Next figure is computed from figure 102 still in case of systematic selection and combination. Compared to the previous example, the difference lies in the order of methods valid at least for *dx* : now, *m4x* precedes *m1x* and *m4xy* precedes *m1xy*.

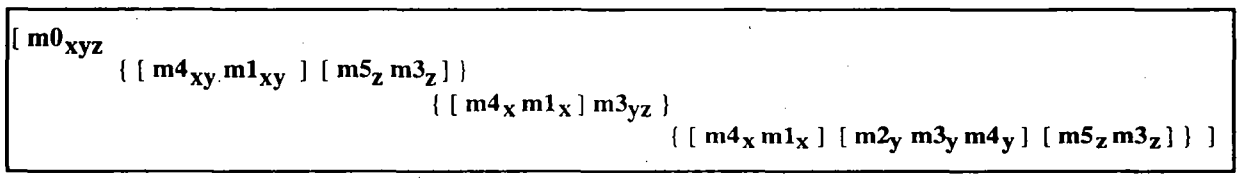
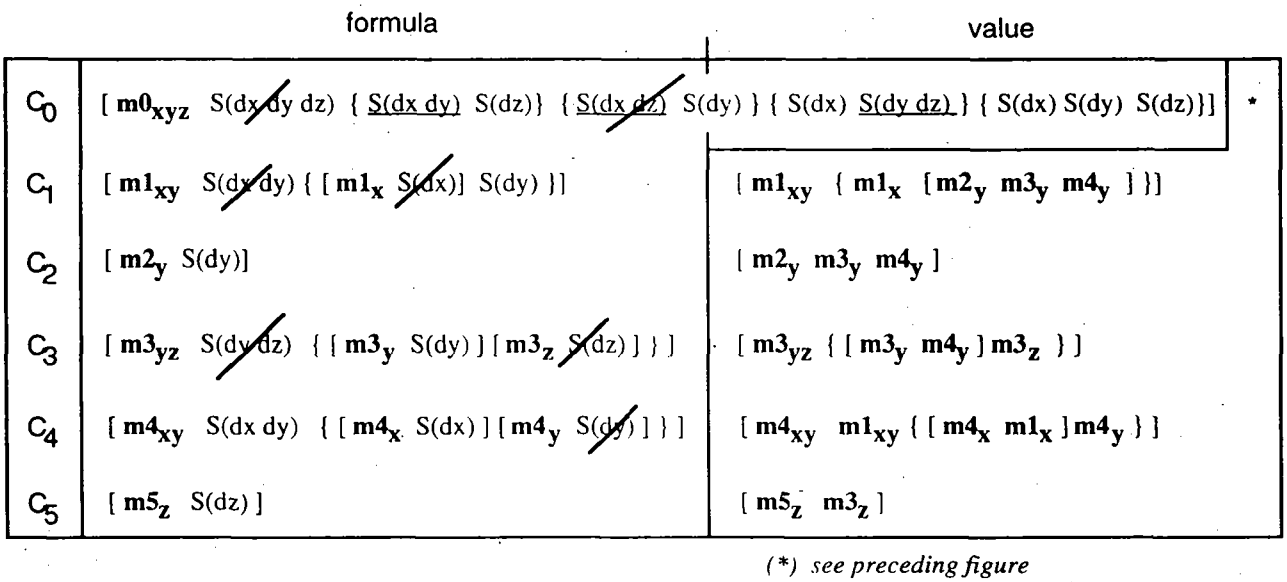


Figure 157.

Next figure shows how the above result can be computed. Note that the paths followed by the *S (dx)* and *S (dx dy)* demands is not the same than for the *S (dy)* demand. The demands *S (dx)* and *S (dx dy)* are answered by *C4* when sent by *C0* ; and by *C1* when sent by *C4*. The demands *S (dy)* is answered by *C1* when sent by *C0* ; by *C2* when sent by *C1* ; by *C3* when sent by *C2* ; and by *C4* when sent by *C3*.



(*) see preceding figure

Figure 158.

b) Regular hierarchy

In this case, we consider figure 106 instead of figure 102. The result is easily obtained from the invocation sequence diagram of figure 108. It also corresponds to the case shown in figure 148.

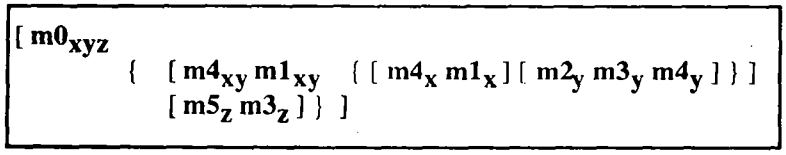


Figure 159.

c) Simplified form

Were the simplified form be used, the combined method will be reduced to $m0_{xyz}$ in any case (no difference with the previous example).

8.3 REGULARITY : ITS CONDITIONS

The regularity of a hierarchy vs. its methods is an important property. It has been defined previously : a hierarchy is **regular** vs. its methods named *m*, if all *m* methods of degree≥ K that exist along a same dimension satisfy the same *K* dimensions (cf. §7.3.2.b.2.1.δ). When a hierarchy is regular, its invocation diagram is a (list of) tree(s). It was also said that regularity was obtained for memory representations given our choices. So let's first prove this point.

8.3.1 Memory representations

A memory representation is a list of cells that can be specified for each color in a c-graph or for each pigment in a p-graph. No memory representation can be specified for a blend : memory representations for blends are computed automatically from memory representations defined for pigments. No memory representation can be specified for abstract dimensions (i.e. in case the sole pigment of the dimension is specified as ":abstract" or ":property", or in case of a mixin).

Because no memory representation can be defined for a blend, the degree of any memory representation defined along a same dimension is one. No interleaving is thus possible. Hence, the expected result.

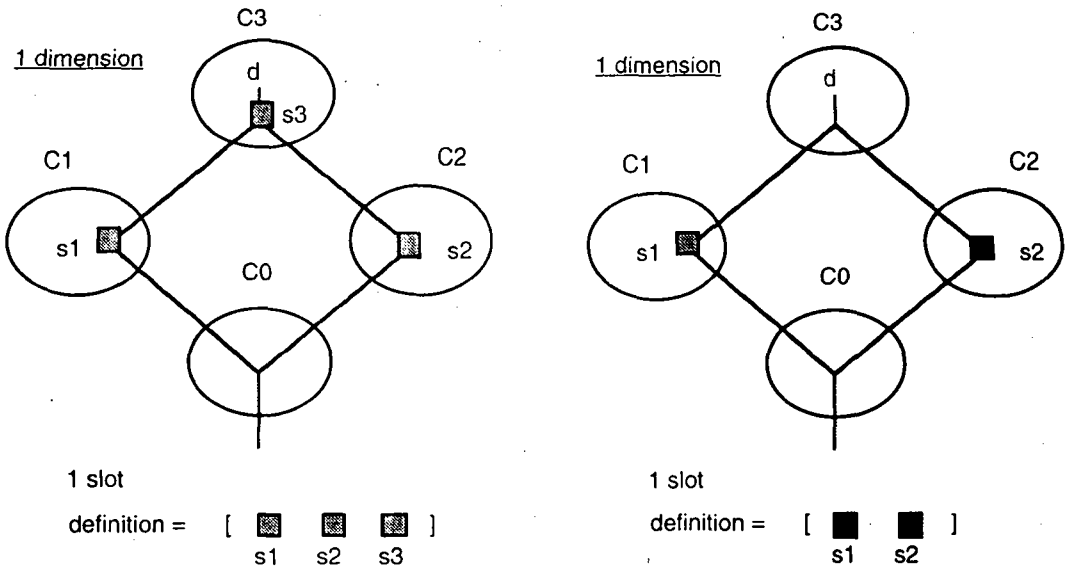
a) Systematic combination (of definitions)

Let's now suppose class *C* specifies a memory representation for a given pigment (or color). In a subclass of *C*, the memory representation for that pigment (same dimension) cannot be replaced. It can only be upgraded :
— if an already existing cell is named, the new specification is combined with the previous specifications of the same cell (as done in CLOS) ;
— if a new cell is defined, this new cell is added to the memory representation of the dimension in question.

Next three figures show, from the point of view of *C0*, what happens when a cell with the same name (be it *s*) is inherited via different paths :

— in the first two figures, the cell in question implements the same dimension : in one case, a first definition of the cell in *C3* is refined independently in *C1* and *C2* ; in the second case, the cell is initially defined independently in *C1* and *C2*. A single cell named *s* exists in *C0, C1, C2 and C3*. In any case, the definition in *C0* combines the inherited definitions in the order defined by the linearization of the hierarchy, i.e. (*C0 C1 C2 C3*) ;

— in the third figure, two dimensions exist that define a cell named *s*. In this case, two cells with the same name *s* are defined. A warning is issued : when updating or consulting a cell named *s*, the user should specify which one is to be used (naming mechanism). An alternative solution is to rename one cell *s*.



Figures 160 & 161.

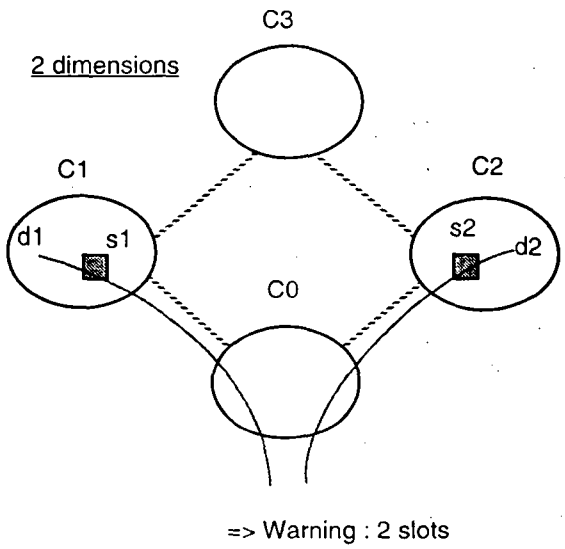


Figure 162.

b) Systematic masking

In case of systematic masking, no combination of definitions is done for a given cell : only the first definition (vs. the order of the linearization) is kept. Cells pertaining to different dimensions should have a different name, otherwise their use should specify which one is to be updated or consulted (naming mechanism).

8.3.2 Methods

Figure 94 shows an example of interleaving dimensions. Regularity does not exist for this hierarchy. As explained in the corresponding text (§7.3.2.b.2.1.δ), this topological case corresponds a priori to the reality. Such a coexistence is however questionable in practical situations.

First of all, let's insist on the meaning of the notation. A method *mxy* is a method that effectively requires both *dx* and *dy* dimensions. For example, to display a *Circle* instance, the *draw* method requires both the *radius* and the *center* : it is strictly impossible to *draw* such an instance by combining two elementary methods, one that would use *radius* and one that would use *center*.

Second important point : it is difficult to imagine that two dimensions that are employed[48] to implement a method *m* of class *C* may be considered apart in a subclass of *C* for implementing the *m* method. Continuing with the same example, if we were to *draw* a *Cylinder* instance, the *radius* and *center* dimensions will remain associated : the result will not be obtained, for example, thanks to a first method requiring both the *center* and the *height*, and a second method requiring only the *radius*.

Absence of regularity thus normally appears as a consequence of the linearization. However, it seems that all potential problems disappear in fact due to explicit masking (this one being set up by the user for semantic reasons). The *STQ* hierarchy provides two examples that exactly follow this pattern :

— as already noted, the *STQ* hierarchy is not regular vs. the *print* methods (see figure 129). Two *print* methods, one inherited from *Stack* (*LIFO*, *bag* and *object* dimensions), one from *Queue* (*FIFO*, *bag* and *object* dimensions), do induce interleaving dimensions after linearization. Yet, the problem is naturally eliminated by the user : since the *STQ* instance is not to be *print*ed twice —once as a *Stack* and once as a *Queue*— but only once, explicit masking is specified ;

— another potential problem is also eliminated naturally, almost silently, concerning *STQ*. This class inherits from two *pop* methods : one from *Stack* (*LIFO* and *bag* dimensions) and one from *Queue* (*FIFO* and *bag* dimensions). Potentially, the *STQ* hierarchy is thus not regular vs. *pop*. Yet, when a *pop* message is issued, a single element is to be removed from the *STQ* instance and not two. Masking is thus used when necessary. In the invocation sequence diagram for state *(b x)* or *(b y)*, the method *pop* (*dequeue*) of *Queue* does not appear (it is masked). In the invocation sequence diagram for state *(a y)*, the method *pop* of *Stack* is not available : only the *pop* method of *Queue* does exist. Hence, no problem actually exists (cf. figure 33 in §5.2.1.a).

Our expectation is thus that a hierarchy is normally regular vs. its methods once methods explicitely masked have been removed.

A hierarchy may also appear non regular in case two methods having no semantic relationship happen to be given the same name (name collision). This is an abnormal condition. The user is warned and invited to rename one or both methods.

Were a hierarchy non regular, method combination is still possible (cf. subsection 8.1.3).

8.4 CONCLUSION

Regularity will thus be supposed for the rest of the paper :

— this is a priori not a big constraint since a hierarchy appears to be semantically unnatural if not regular ;

— this is advantageous : the systematic combination of items is much easier to deal with when hierarchies are regular.

[48] Note the difference betwen "involved" and "employed" : in the *STQ* example, the *object* dimension is the only one involved ; the *bag* dimension is employed, yet not involved.

9. MULTIPLE DISPATCH

Historically, one generalization of OOP was the passage from the concept of "message" sent to a "receiver" (discrimination done according to a single argument, the first one) to the concept of "generic function" call (discrimination done according to several arguments). In the first case (single dispatch), methods are attached to a single class and apply solely to objects that inherit from that class. In the second case (multiple dispatch), a method is a priori non longer attached to one class, but is associated to several ones, termed "specializers", according to its list of required parameters[49] ; this method applies when the (required) arguments of a (generic function) call respectively match their "specializers".

Until now, the paper was focusing on single dispatch. Yet, as OOP, COP can be generalized to multiple dispatch too. Instead of transitions strictly defined inside one color graph, **multi-transitions** are considered (cf. subsection 2.1 in companion paper 1) : these involved several color graphs. Thus, color graphs may now be coupled instead of being systematically standing alone as in the case of single dispatch.

Computing the effective method to be run when a (generic function) call is issued comprises four parts : (a) satisfying a multi-transition ; (b) selecting methods ; (c) checking that these methods are valid ; (4) producing the combined method. As a matter of fact, these steps usually corresponds to a systematic combination style (cf. CLOS). Since our text also envisages a systematic masking style, we first consider this context.

9.1 SYSTEMATIC MASKING STYLE

Suppose a (generic function) call *m* is made with a number of (required) arguments : the first argument, an instance of class C_{X0}, is in a given state *xo* (possibly expressed using a set of pigments pi_{X0}) ; the second argument, an instance of class C_{Y0}, is in state *yo* (possibly expressed using a set of pigments pj_{Y0}) ; etc. Be *x*, *y*, ... these objects. The search for an effective *m* method generalizes the search done for one argument.

9.1.1 Satisfying a multi-transition

First, the (generic function) call must be acceptable : a **multi-transition** must be found that satisfies all (required) arguments. Each (required) argument should meet (vs. the specification of the corresponding parameter) the two conditions to be met by the first argument in case of a message (vs. the specification of the first parameter) : (1) each argument *x* (resp. *y*, ...) must be an instance of the first (resp. second,...) specializer of the multi-transition or the argument class must be a subclass of this specializer (same condition than in CLOS) ; (2) the state of the argument should match (one of) the state(s) specified with the specializer.
In other words, given the positions of the (mini-)tokens in each color graph of classes C_{X0}, C_{Y0}, ..., a multi-transition *m* is searched going up in the ancestor-trees of *x0*, *y0*, ... : if not found in C_{X0}, C_{Y0}, ..., the search recursively continues in the ancestors of C_{X0}, C_{Y0}, ... until found. If no multi-transition is found, an error is signalled.

[Since this subsection especially addresses the case where several color graphs are coupled via the *m* transition, we restrict the following discussion to the case where more than one color graph are involved ; otherwise, all happens as in the case of a message.]

9.1.2 Selecting methods

The obtained *m* multi-transition determines which dimensions *dx1*, *dx2*, ... (resp. *dy1*, *dy2*,...) of the *x* (resp.*y* instance) are involved. If the algorithm for searching a method in case of a message approach (recursive "p-graph local search and combination" rule along the involved dimensions) is used on the first argument *x*, this determines a set of micro-methods satisfying the involved dimensions (*dx1*, *dx2*,...) as well as a number of employed -yet not involved- dimensions of *CX0*. However, the methods that are found may well not satisfy the other arguments.

Next figure shows an example : concerning the instance *x*, the involved dimensions for *m* are supposed to be *dx1* and *dx2* ; concerning *y*, they are supposed to be *dy1* and *dy2*. Running the algorithm for the sole *x* argument yields methods *mxy1* and *mx2* : *mxy1* satisfies *dx1* (involved dimension) as well as *ex1* (employed dimension) and *dy1* (involved

[49] Here, "parameter" means "formal argument". And "argument" means "actual argument". The expression "(required) parameter" refers to the CLOS specification featuring other sorts of parameters (optional, keywords,...). Only required parameters may be specialized.

dimension) ; *mx2* satisfies only *dx2*. This set of micro-methods does not satisfy the *x* and *y* arguments since *dy2* is not satisfied.

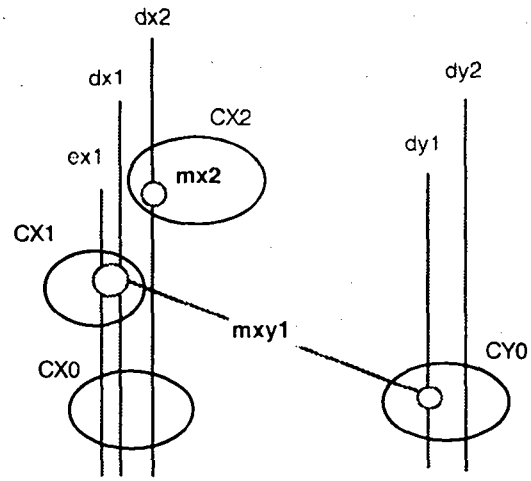


Figure 163.

Hence, the following adaptation. The "p-ancestor-tree local search" rule is run for *dx1* (the first involved dimension of the first argument). If a method *mx* is found, this one satisfies *dx1* and —possibly— a number of other dimensions of the first, second, etc. arguments (involved or only employed). If *mx* does not satisfy one or more involved dimensions for each other argument, the *mx* method is not retained. The "p-ancestor-tree local search" rule is run again for *dx1*, starting the search above *mx*. The same process is run until a method *mxy...1* is found that satisfies at least one dimension of each other argument (*y*, z,...). The search may also stop because no more method can be found along the *dx1* dimension : in this case, an error is signalled.

So, let's suppose a *mxy...1* method has been found. It satisfies *dx1* and —possibly— a number of other dimensions of the first argument (involved or only employed dimensions). The above adapted algorithm is then run on the next unsatisfied involved dimension of this first argument, be it *dxk*. Hence, either an error or the finding of a method *mxy...k*. This method *mxy...k* satisfies *dxk* and —possibly— a number of other dimensions of the first argument (involved or only employed dimensions). If all the involved *dxi* dimensions are satisfied, then we are done. Otherwise, the above adapted algorithm is run again, on the (new) next unsatisfied involved dimension... This process is repeated until an error is signalled or all the involved *dxi* dimensions are satisfied.

The set of methods that is found satisfies by construction all the involved *dxi* dimensions (*dx1*, *dx2*, ...) once and only once.

9.1.3 Checking the selected methods

The set of methods that is found are multi-methods in the sense that they require several parameters. They should not only satisfy (once and only once) the involved dimensions for the first argument, but also those of the second, third, etc. arguments. They should also satisfy once and only once all the employed -yet not involved- dimensions for the first, second, third, etc. arguments.

Next figure shows a counter-example. Concerning the *x* argument, the solution is to be rejected since the employed dimension *ex3* is satisfied twice, once by *mxy3* and once by *mxy5*. Concerning the *y* argument, the solution is also to be rejected. Three reasons : (a) the involved dimension *dy5* is not satisfied ; (b) the (involved or only employed) dimension *dy3* is satisfied twice (once by *mxy3*, once by *mxy4*) ; (c) the involved dimension *dy1* is satisfied twice (once by *mxy1*, once by *mxy4*). Note this last configuration cannot appear in a message approach, i.e. when but one argument is considered for dispatching.

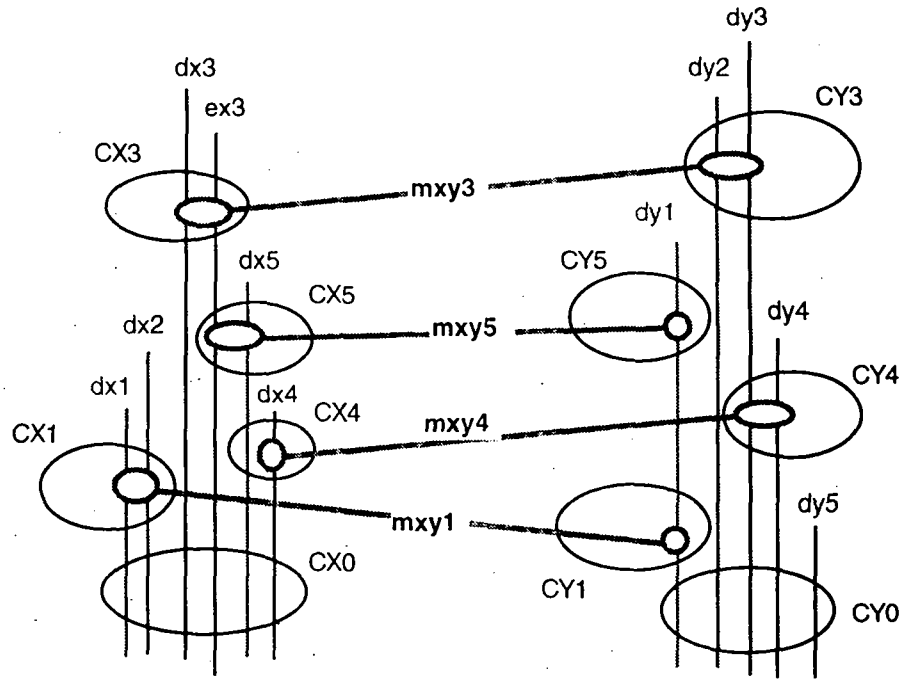


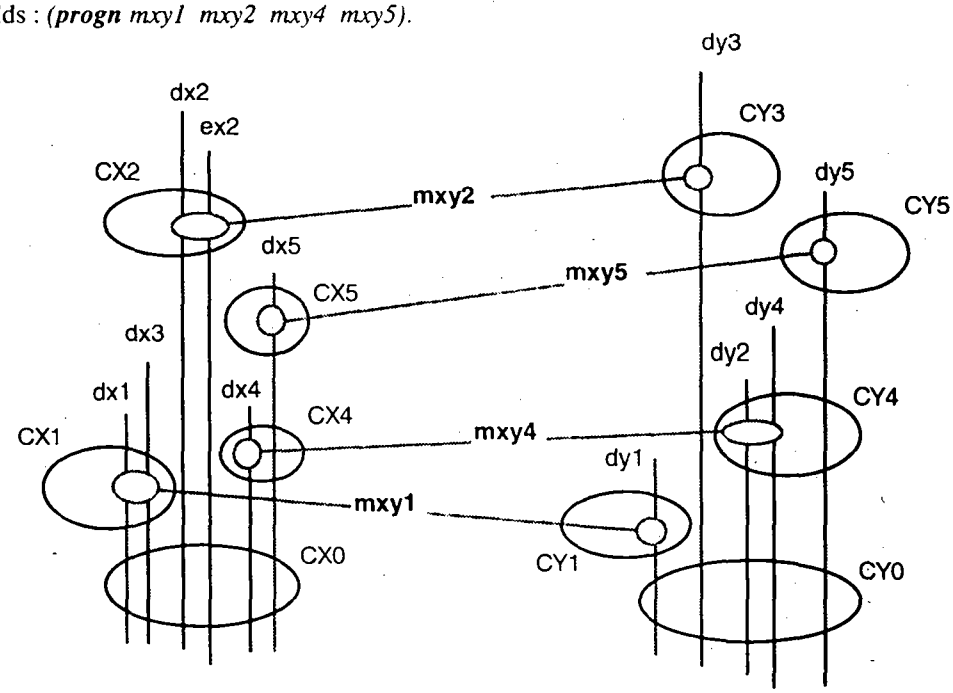
Figure 164.

All checkings isolated here for explanation purpose can be done incrementaly each time a new method is selected.

9.1.4 Producing the combined method

The order of the selected methods for the dimensions *dyi*, *dzi*,... (of the second, third,... arguments) is determined by the order chosen for the dimensions *dxi* (of the first argument). Thus all happens as if we were combining micro-methods in case of single dispatch (i.e. in case of a message with argument x). The usual rule thus applies : rephrased, it says that the micro-methods are to be combined in the order they satisfy their leftmost involved dimension.

Next figure shows an example : *mxy1* is chosen first since it satisfies the first dimension *dx1* ; *mxy2* is then chosen since it satisfies the second dimension *dx2* ; *dx3* is already satisfied by *mxy1*, so no method selection is done for it ; *mxy4* is then chosen before *mxy5* (due to *dx4* and finally *dx5*). Hence the result below. If the default {combination} method is chosen, this yields : ***(progn** mxy1 mxy2 mxy4 mxy5)*.



combined method => { mxy1 mxy2 mxy4 mxy5 }

Figure 165.

9.2 SYSTEMATIC COMBINATION STYLE

In case of message (single dispatch), systematic combination of methods was proposed under the hypothesis of regularity (cf. §7.3.2.b.2.1.δ). For multiple dispatch, we consider an extension of this hypothesis which we call **global regularity**. In the following, we first characterize this property before expressing the combination algorithm. (Note that the "prevalence of combined items" rule is systematically applied.)

9.2.1 Global regularity

a) Definition

Regularity was previously defined for one given hierarchy vs. its *m* methods : *"all m methods of degree≥ K that exist along a same dimension should satisfy the same K dimensions"*. Instead of a single hierarchy rooted in *C0* (receiver's class), we now have to take into account one hierarchy per required argument. The criteria we propose simply extends the rule to all *dxi*, *dyi*, ... (involved and employed) dimensions of the hierarchies rooted in C_{X0}, C_{Y0},... (the classes of the arguments). Global regularity means the checking takes into account all *dxi*, *dyi*, ... dimensions not separately but at the same time.

Next figure shows a counter-example : each hierarchy is obviously regular when considered alone ; yet, global regularity is not obtained : along the *dx1* dimensions, the dimensions that are satisfied by *mxy1* are *dx1* and *dy1* (degree 2), whereas the dimensions satisfied by *mxy2* are *dx1* and *dy2* (degree 2). Thus, the criteria for global regularity is not verified (two methods of degree 2 do not satisfy the two same dimensions).

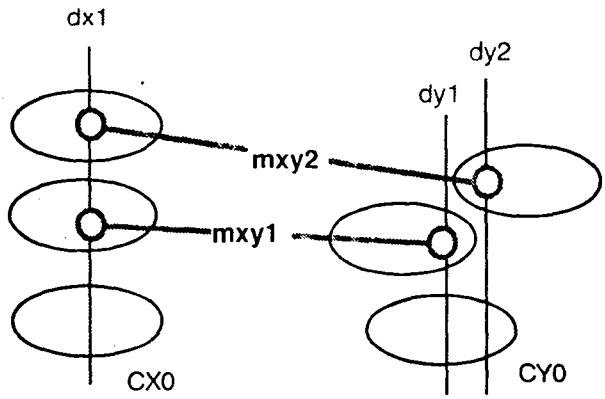


Figure 166.

b) Properties

Due to global regularity, a number of topological properties can be stated :

— the dimensions *dxi*, *dyj*,...are aggregated in stable ways : for example, *dx1* is always associated with *dy2*, *dx2* and *dx3* with *dy3* and *dy5*, *dx4* with *dy4* and *dy6*. Any tree made by a group *dxi* is associated to a tree, always the same, made by a group *dyj*, ... ;

— the trees formed by a group *dxi*, a group *dyj*,... in a same aggregate are isomorphic : if *dx1* is associated to *dy1* and *dx2* to *dy2*, then if *dx1* gets grouped with *dx2*, *dy1* gets grouped with *dy2*. Since the number of dimensions in each group may be different, the alleys (i.e. trunk and branches) of one tree may be larger than the corresponding alleys of another tree in the same aggregate. However, the form of the two trees are identical ;

— the number of nodes on an alley *dxi*... and on an alley *dyj*... that are in correspondance (in two trees of a same agrregate) is identical ;

— if methods are represented by lines joining the nodes they associate, crossing may only exist between nodes on alleys that are in correspondance ;

Next figure illustrates these properties. It shows a single aggregate (there may be several ones in parallel with this one). Here, two groups of dimensions (*dx1* and *dx2* on one side ; *dy1*, *dy2*, and *dy3* on the other side) are always associated. The dimensions *dx1* and *dx2* form a tree ; the dimensions *dy1*, *dy2*, and *dy3* also form a tree. These two trees are isomorphic : they both have three alleys (a trunk and two branches). The number of nodes on each alley (trunk included)

is the same in the two trees (two on each trunk, two on each left branch, one on each right branch). There is one crossing due to methods *m4* and *m5* : the classes of their arguments are not found in the same order in the *CX0* and *CY0* hierarchies.

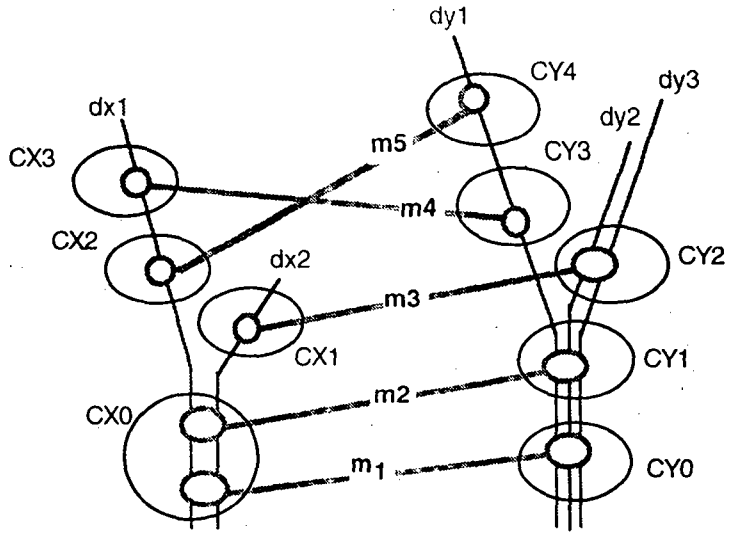


Figure 167.

9.2.2 Selecting methods

Assuming global regularity, search may be organized per group of dimensions. The first solution (obtained by the selection algorithm shown in subsection 9.1.2) enables the identification of one or several groups of *dxi* dimensions. In each group, the search may be done going up in the leftmost *dxi* dimension. If the method found does not satisfy the whole group of dimensions (i.e. if its degree is smaller than the number of the dimensions in question), this is because this group has been fragmented : a search is thus also made along the leftmost unsatisfied dimension. If the method found does not completely satisfy the group of formerly unsatisfied dimensions, a new search is made in the new group of still unsatisfied dimensions. Etc. Once each alley has been identified, a search is done upwards in each one to find new methods along these alleys. And so on.

(Since the selection is done along the *dxi* dimensions in the *CX0* hierarchy, the "prevalence of combined items" rule is likely to be not verified. Thus alleys may appear in disorder compared to the (final) invocation sequence tree. In practice, after fragmentation of the *dxi* dimensions, a new method may regroup one or several subgroups of *dxi*. : this occurs when the method degree is greater than the number of dimensions of the group (subgroup, in fact) to which it was initially thought to be attached.]

9.2.3 Checking the selected methods

During the selection, we have to check that :
— the hierarchies rooted in C_{X0}, C_{Y0}, ... are globally regular vs the *m* methods (cf. subsection 9.2.1) ;
— the set of involved (and -if any- employed) dimensions for the first, second, ... arguments are satisfied once and only once (cf. subsection 9.1.3).

9.2.4 Producing the combined method

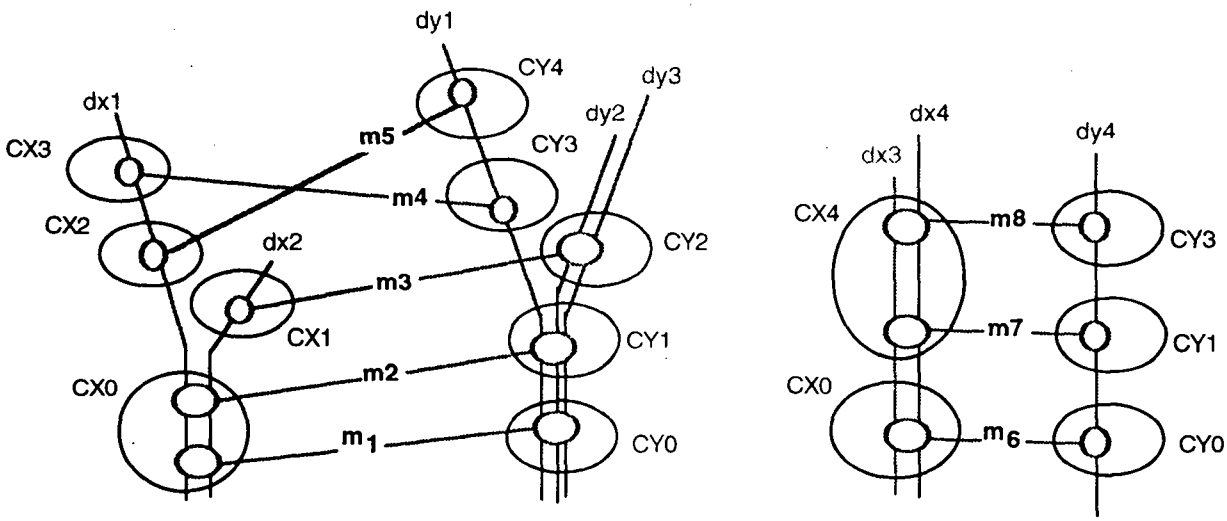
a) Ordering

The methods previously selected are sorted so as to respect the "prevalence of combined items" rule (which is likely to be not observed when the selection is done along the *dxi* dimensions of the *CX0* hierarchy). In other words, the invocation sequence diagram is organized to be a list of trees.

A further re-ordering may be necessary in case several methods are proposed by a same class in the first hierarchy (the one rooted in *CX0*) : in this case, these methods are sorted according to their order vs. the *dyi* dimensions, and if equal in that dimension too, according to their order vs. the *dzk* dimensions, etc. In the previous figure, this case is illustrated by *m1* and *m2* : *m1* is placed before *m2* due to the respective order of the *m1* and *m2* specializers (*CY0* is placed before *CY1*).

b) Combination

Once the ordering of methods has been determined for the *dxi* dimensions, then the *m* methods are ordered. What remains to be done is simply to produce a combined method. As noted for the systematic masking style (subsection 9.1.4), all happens as if we were combining micro-methods in case of single dispatch. Thus, the combined method is produced exactly as if *m* was a message, the *x* instance being the receiver (see section 8). Next figure illustrates this.



combined method => { [m1 m2 { [m5 m4] m3 }] [m6 m7 m8] }

Figure 168.

This combination algorithm may be extended with qualifiers and can be tuned or changed to some extent via MOPs. It generalizes :

- the CLOS algorithm (see next section) : this one corresponds to having a single dimension per argument. Note the ordering of several methods when originating from the same class *X* is also determined in CLOS by looking at the ordering vs. the other arguments ("*To compare the precedence of two [applicable] methiods, their parameter specalizers are examined in order (...) the first pair of parameter specializers that are not equal determines the precedence*" ([Bobrow et alii, 1988], p. I-27) ;
- the message case (one argument) with systematic combination (see section 8) : it corresponds to considering but one argument (the first one, i.e. receiver).
- the generic function case (several arguments) with systematic masking (see subsection 9.1).

10. SOPHISTICATED COMBINATION METHODS WITHOUT INDIVIDUAL METHODS NOR THE SEND-SUPER ANTIMODULAR CONSTRUCT

10.1 GETTING RID OF THE SEND-SUPER CONSTRUCT

10.1.1 A reverse approach

Till section 8, we considered a style where masking was systematic. To somehow examplify this style, we named it "Smalltalk-like". As a matter of fact, masking is not systematic in Smalltalk : sending a message to **super** instead of **self** breaks the masking. Contrary to Smalltalk, CLOS is based on a systematic combination of methods. To be schematic, all methods found in the hierarchy of a class are to be combined. As for Smalltalk, this statement is approximate : if we consider the standard combination method, the **:before** and **:after** methods are all combined ; but the primary and **:around** methods are not. These methods are usually not combined and the approach is similar to Smalltalk : like the **super** mechanism of Smalltalk, an exception mechanism is provided (**call-next-method**). We can summarize this situation in saying that the combination of all methods found in a hierarchy is the rule... except that (for some type of methods the rule is to select one method only except that ... (combination is still possible via the **call-next-method**)).

We propose a more systematic approach : systematic combination unless refused. Note the philosophy is quite different from the Smalltalk style and from the CLOS style (restricted to primary and **:around** methods). The default in these languages is no combination unless an explicit call is made (via **super** or **call-next-method**, generically termed here **send-super**[50]). Our proposal considers the opposite default : the combination is systematic unless an explicit indication is given.

10.1.2 Advantages

This reverse approach has several adavantages.

a) A unifying mechanism

The mechanism we propose is a unifying vs. the Smalltalk and CLOS styles :

— to emulate a Smalltalk-style, the indication 'masking should occur' is basically set for every micro-method. When a Smalltalk-method uses **super**, its equivalent micro-methods are not attached the indication ;

— to emulate a CLOS-style, the micro-methods equivalent to the **:before** and **:after** methods are not attached the indication ; those equivalent to the primary and **:around** methods are attached the indication under conditions similar to Smalltalk (i.e. when **call next-method** does not appear inside the CLOS methods in question).

b) A purely declarative mechanism

A question which is somewhat perturbing in traditional OOP is that it mingles declarative and imperative programming techniques for method combination.

The declarative technique consists in attaching qualifiers (like **:before**, or **:after**) to method declarations and using these qualifiers as a basis for method combination, notably for implementing MOPs : present in CLOS, this technique is (unfortunately) not quite frequently supported by OO languages.

To be general, the imperative technique is due to the OO construct used for calling, from inside a given method body, a method having a same name but normally masked (for example, because it is defined in a superclass of the class in which this method body exists). Such a construct (ex. : **super** in Smalltalk ; **call-next-method** in CLOS) allows reuse while preventing infinite looping, but it is non modular. Sonya Keene, a member of the CLOS committee, expresses her worry about it in clear terms[51]. While the declarative technique lets the user *"predict the order of methods without looking at the code in the bodies of the methods"*, the imperative technique is *"in a sense (...) a violation of modularity"* and should

---

[50] **send-super** is suggestive but not quite good from a semantic point of view since the next method which is called is determined vs. the linearized list of classes (termed class precedence list in CLOS) and is thus not necessarily in an ancestor class of the class possessing the method from which the **send-super** is issued : **call-next-method**, from this point of view, is better. Yet, because our proposition is general and not directed against **call-next-method** in particular, we prefer to use **send-super**.

[51] [Keene, 1989], section 5.8, p.111 : *"Guidelines on controlling the generic dispatch"*.

thus be used *"only when that power is truly necessary"*. Unfortunately, *"some programs"* cannot be written*"without resorting to the imperative technique"*.

As shown below, our proposal consists in attaching to the header of micro-methods (or, possibly, transitions) the indication 'masking should occur' in the form of a keyword (**:masking**), exactly like the **:before** or **:after** keywords. As shown below, no **call-next-method** (nor **super**) is any longer necessary.

10.2. GETTING RID OF INDIVIDUAL METHODS

In CLOS, a parameter specializer may either be a class or a list (**eql** *form*), *form* being evaluated once, at the time the method is defined. Be *object* the result of evaluating the *form* : (**eql** *form*) denotes in fact (**eql** *object*). This feature enables a filtering on a particular object. For example, if we want to implement the *factorial* method, we can write a general method using **Integer** as a specializer and a specific method using (**eql** 0) to stop the recursion. (This style of programming, without any explicit testing, illustrates what is done in COP.)

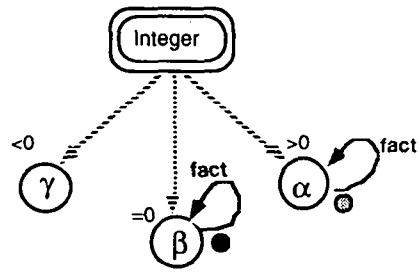
```
(defgeneric    fact (x))

(defmethod   fact ( (x   Integer ))
      (* x (fact (- x 1))))

(defmethod   fact ( (x   (eql 0) ))
      1)
```

Figure 169.

In COP, there is no need for an (eql form), since an object may be identified with a particular state. Next two figures show this for the *factorial* example, both in visual and textual form. (Note that we made the code more secure by distinguishing the negative integers and avoiding to attach them the *factorial* transition.)




```
(deftransition      fact ( (x   Integer (  α α) ( β β)))

(defmethod   fact ( (x   Integer   α))
      (* x (fact (- x 1))))

(defmethod   fact ( (x   Integer   β))
      1)
```

Figures 170 & 171.

10.3 SOPHISTICATED COMBINATIONS

As annouced before, our basic scheme can effectively be tuned so as to mimick –without the harmful **send-super** construct– the mechanisms supported by traditional OOP. As an example, we will consider the sophisticated CLOS ones. The goal is to let the user systematically *"predict the order of methods without looking at the code in the bodies of the methods"* without resorting to **call-next-method** and its companion **next-method-p** (which tests within the body of a method if a next method exists).

10.3.1 Basic idea

First, let's review the different cases of method combinations in CLOS supposing a single dimension per class hierarchy to ease the initial comparison. (In the next subsection, this restriction will be removed.)

a) The standard method combination

This frequent combination deals with **:before**, **:after**, primary and **:around** methods. Original methods of a CLOS program will generally be splitted up according to the class colors and pigments. Certain **call-next-method** invocations may vanish in the process.

The solution we propose applies to the bodies of the resulting methods :
- **:before** methods are represented as pre-methods, **:after** methods as post-methods ;
- primary and **:around** methods are represented as pairs of pre- and post-methods.

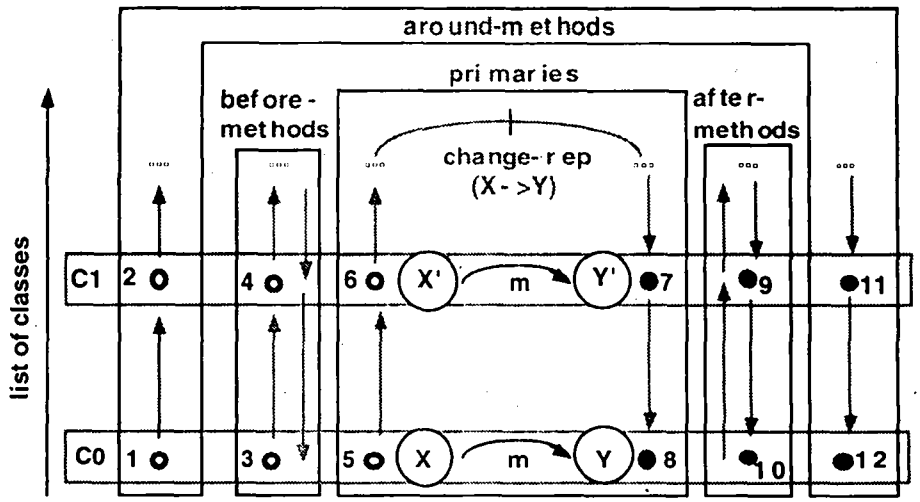


Figure 172.

The above figure illustrates our proposal (the numbers give the order of method executions).

Note the relative ordering of methods in CLOS (ex. : the running of **:before** methods from most specific to least specific or the running of **:after** methods from least specific to most specific) appears as a mere consequence of our more general mechanism (pre-methods in ascending order, post-methods in descending order).

Extra data control may be provided to respect in details the **call-next-method** semantics[52]. (The consequence of method splitting about temporaries is also dealt with[53].)

b) Other method combinations

Besides the standard method combination, CLOS offers a number of other built-in method combination types (simple ones, like **progn** or **max**). In these, the **:around** and primary roles are recognized. The former role may use **call-next-method** (and **next-method-p**) exactly as in the standard method combination : the above analysis[54] thus applies. The latter role can't use **call-next-method** (hence, a simpler treatment than with the standard method combination) ; order of primaries is defaulted to **:most-specific-first** but can be changed to **:most-specific-last** : obviously, such an effect can easily be obtained in our approach by specifying the primary methods as being pre- or post-methods.

A user is also given two possibilities to define other method combination types. The first one, termed "short form", leads to the considerations just evoked (the simple built-in method combination types act as if they were defined using this form). The second one, termed "long form", provides great flexibility. From the perspective of the current discussion, this form leads to the considerations already evoked, either about the standard method combination type (which can be defined using it) or about order of methods.

---

52 - **to-next-method** is reserved to exceptionally transmit explicit arguments from a primary or **:around** micro-method to the next micro-method (upwards), i.e. from a pre-method to the next one. (In any case, a pre- and its associated post-method receives the same arguments.) Using **to-next-method** when no next pre-method exists is an error ;

- **from-previous-method** is reserved to transmit the result(s) of a less specific primary or **:around** micro-method to its caller, i.e. the result(s) of a given post-method to the next post-method. The returned results, if necessary, are to be bound to (at least) an argument or a temporary. Using **from-previous-method** when no previous post-method exists is an error.

53 As mentionned before, a **shared** declaration will be used to transmit data (temporaries) from a pre-method to its associated post-method, thus mimicking temporaries shared thru a common encircling **let** in CLOS. (An efficient implementation is possible.)

54 Primary or **:around** methods are typically made of a first part making a number of computations, followed by a **call-next-method** the result(s) of which is (are) used in a second part. The pre-method corresponds to the first part ; the post-method, to the second. More complex schemes may appear in traditional OOP (presence of several instances of **call-next-method** instead of a single one, possibly under the control of test or loop statements). In the COP view, these extra controls are obtained using colors at an appropriate level.

10.3.2 Generalization

a) Single dispatch

a.1) Justification

The invocation sequence diagram of a regular hierarchy is a tree or a list of trees, each one being made by a group of one or several dimensions. Note that the combined method which is obtained, say for for pre-methods, was established without requiring the [combination] to be specified in detail. Same remark for the second {combination}.

In case of an isolated dimension, the tree is a simple list. Thus, the above proposition applies directly : the list of classes to be considered is the one obtained by linearization ; all methods are ordered along this line ; pre-methods are combined in ascending order and post-methods in descending order. (Note this choice exactly induces the CLOS ordering[55] for **:before**, **:after**, **:around** and primaries without requiring any further specification.)
As a matter of fact, the line constitutes by itself an alley. In the [combination] which is obtained, items are used but once and the way they intervene is a priori compatible with an ascending order (i.e. from most specific to least specific) or a descending order (i.e. from least specific to most specific) : the interpretation of the [combination] may be different in case of pre-methods (ascending order is the default : methods are processed from left to right between the brackets) and post-methods (descending order is the default : methods are processed from right to left between the brackets). Hence, the compatibility with of the CLOS combination.

In case of a group of cooperative dimensions, the set of methods to be considered are those that are valid for the group : the above figure does not hold any longer as such since methods are along several lines instead of a single one, these lines forming a tree. However :

— in case of a degenerated tree (a list), there is no difference with the case of an isolated dimension : the ordering of methods is like the CLOS one. (All methods have the same degree, otherwise dimensions would not be organized as a list, but as a tree. Because these methods have the same de gree, the prevalence of combined items rule was not applied and thus has not introduced any perturbation in their initial order —the linearization order. Say in a different way, all dimensions except one may be conceptually removed : this does not perturb the order of the methods.) ;

— in case of a non degenerated tree, what has been said for an isolated dimension is also valid for the items along each alley. Since the set of alleys form a tree, the result for an alley can be extended to the whole tree : any item is used but once and the way it intervenes is a priori compatible with an ascending order or a descending order. Thus, in this case, our proposal generalizes the CLOS combination .

Concerning the {combination}, mostly used in a tree but also for collecting the combined method of each group), being not constrained in all its details, it allows plenty of variations. Parallel computations are possible. Otherwise, computations on a left alley may be done before or after the computations on a right alley. The first option is the most natural one since it respects the order of the dimensions, which is usual in COP. The choice also impacts the result returned by the combined method. The proposed default is thus a processing from left to right.

Let's now suppose that both pre- and post-methods exist along the *dxi* dimensions. Suppose, for simplicity, that the invocation sequence diagram is made of only two isolated dimensions, *dx0* and *dx1*. Since dimensions are independent, there is no difference between the following orders : (a) ascending *dx0*, then *dx1* (pre-methods), then descending *dx0*, and finally *dx1* (post-methods) ; (b) ascending and descending *dx0* (pre- and post-methods of *dx0*), then ascending and descending *dx1* (pre- and post-methods of *dx1*). This result can be extended to diverging alleys in a same tree. Thus, we can process each alley in turn for both the pre-and post-methods, or each group in turn for both the pre-and post-methods, or process the whole invocation sequence diagram for pre-methods, then for post-methods. The result would be the same in any case (if the whole sequence is not interrupted). The proposed default is the latter case, but this can be changed (MOP).

This can be generalized in case qualifiers are used for enabling sophisticated combination methods, so as to mimick CLOS (cf. figure 172). We can either process the whole invocation sequence diagram (default), or each group or each alley in turn, first for the pre-methods (in order : around, before, primary methods) and then for the post-methods (in order : primary, after, around methods). Mixed solutions are also possible.

In any case, this generalizes appropriately the CLOS combination along the list of classes obtained after linearization in the restricted case of a single specializer (single dispatch). Were the invocation sequence diagram reduced to a single degenerated tree (one or several parallel dimensions), the result would be identical to CLOS.

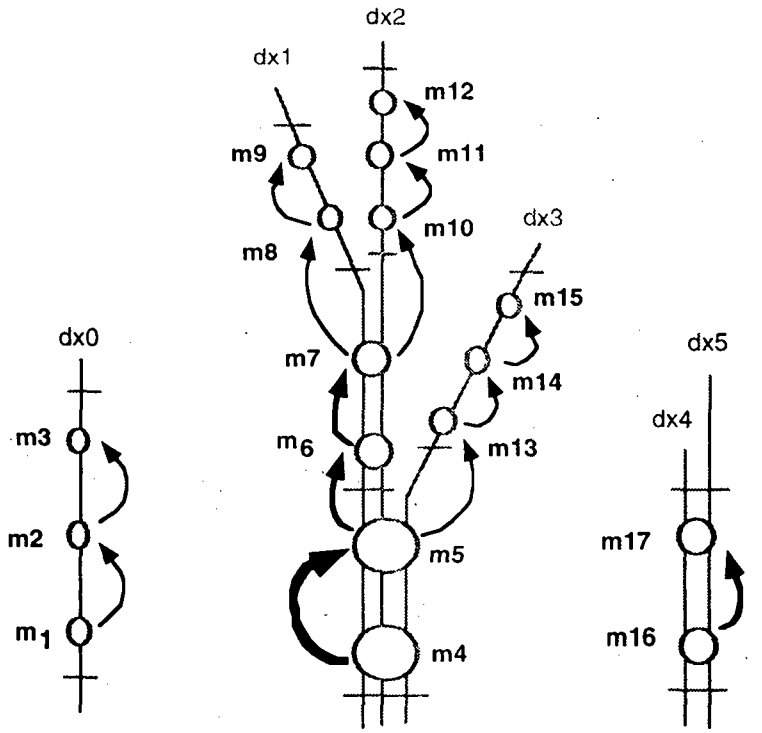
---

55 When comparing an ordering of methods with the ordering that would be obtained in CLOS, we implicitely suppose the linearization algorithm to be the same (the CLOS linearization algorithm is replaced by ours which is monotonic).

a.2) Example 1 : pre-methods

Next figure shows the invocation sequence diagram of a regular hierarchy. Supposing that the bubbles represent pre-methods, the ascending order is represented using arrows. Let's focus on each group of dimensions in turn :
— the first dimension *dx0* is isolated : if it were alone, the combination we propose would be identical to the CLOS one ;
— the next three dimensions *dx1*, *dx2* and *dx3* are cooperative. They form a tree. As explained above, the combination we propose generalizes the CLOS one ;
— the last two dimensions *dx4* and *dx5* are cooperative. The tree they form is degenerated into a list. Thus, the methods along these sole dimensions combine exactly like in CLOS.

In the combined method established for the whole invocation sequence diagram, pre-methods appear in the order of a bottom-up, left to right, pre-order traversal.




```
combined method => {   [ m1 m2 m3   ]
                       [ m4 m5   { [ m6 m7   { [ m8 m9 ] [ m10 m11 m12   ] } ]
                                   [ m13 m14 m15    ] } ]
                       [m16 m17  ] }
```

Figure 173.

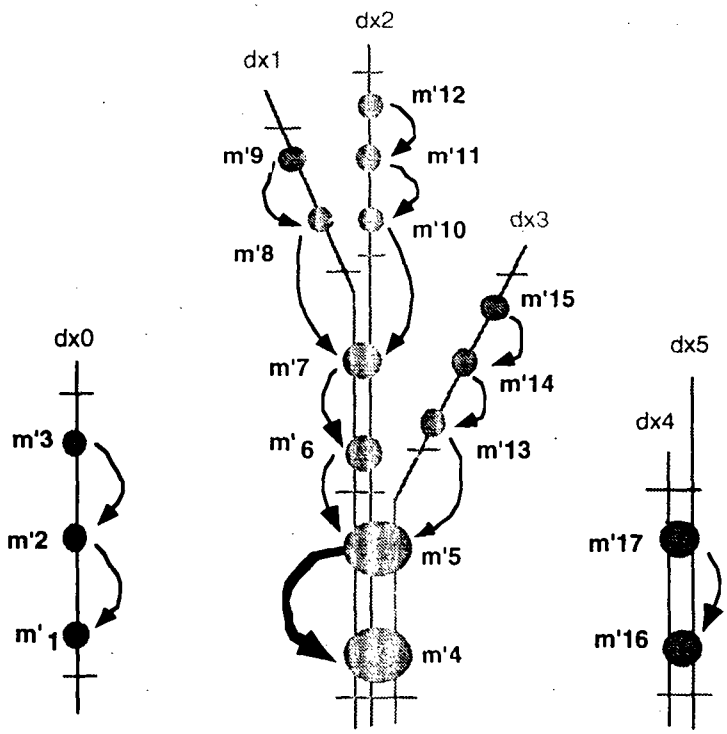
This figure is a priori valid for :**before** methods as well as for pre-methods of primaries or **:around** methods (masked methods are supposed to have been filtered out before the systematic selection and combination). Thus, if we introduce the appropriate qualifiers, three combined methods may be obtained using the above formula (say, *before-combined*, *primary-p-combined*, *around-p-combined*.

If :**before** methods, pre-methods of primaries and pre-methods of **:around** methods are all to be run, one solution is, of course, to run *around-p-combined*, *before-combined* and *primary-p-combined* in that order. Yet, since each alley is independent, two other notable solutions are possible : (1) take each group (list or tree) in turn, and for each one, produce the primary, the after and the around combined methods ; (2) take each alley in turn, and for each one, produce the primary, the after and the around combined methods. Mixed solutions are also possible. In any case, this corresponds to what is done in CLOS (see figure 172) with the methods numbered 1, 2,... (*around-p-combined*), then 3, 4,... (*before-combined*) and finally 5, 6,... (*primary-p-combined*). Were the invocation sequence diagram reduced to a single degenerated tree (one or several parallel dimensions), the result would be identical to CLOS.

a.3) Example 2 : post-methods

Next figure examplifies the case of an invocation state diagram with post-methods only : each *m'...* method is a post-method ; the descending order is represented by arrows.

Post-methods appear in the combined method for the whole invocation state diagram as found in a post-order, top-down, left to right traversal. It is easy to check that the same formula can be obtained from the one established for pre-methods simply by inverting the order of methods or {combined} methods inside each pair of brackets (and changing *m...* to *m'...*). In other words, as already mentionned above, the same formula can be used for pre- and pos-methods, with simply a different interpretation of the [combination] in each case : from left to right in case of pre-methods ; from right to left in case of post-methods.



combined method => { [m'3 m'2 m'1]
[{ [{ [m'9 m'8] [m'12 m'11 m'10] } m'7 m'6]
[m'15 m'14 m'13] }
m'5 m'4]
[m'17 m'16] }

Figure 174.

The figure is a priori valid for :**after** methods as well as for post-methods of primaries or **:around** methods (masked methods are supposed to have been filtered out before the systematic selection and combination). Thus, if we introduce the appropriate qualifiers, three combined methods may be obtained using the above formula (say, *after-combined, primary-P-combined, around-P-combined.*

If :**after** methods, post-methods of primaries and post-methods of **:around** methods are all to be run, one solution is, of course, to run *primary-P-combined, after-combined, around-P-combined.* Yet, since each alley is independent, two other notable solutions are possible : (1) take each group (list or tree) in turn, and for each one, produce the primary, the after and the around combined methods ; (2) take each alley in turn, and for each one, produce the primary, the after and the around combined methods. Mixed solutions are also possible. In any case, this corresponds to what is done in CLOS (see figure 172) with the methods numbered 7, 8,... (*primary-P-combined*), then 9, 10,... (*after-combined*) and finally 11, 12,... (*around-P-combined*). Were the invocation sequence diagram reduced to a single degenerated tree (one or several parallel dimensions), the result would be identical to CLOS.

a.4) Example 3 : pre- and post-methods

Next figure examplifies the case of an invocation state diagram with a set of pre- and post-methods : each *m...* method is a pre-method ; each *m'...* method is a post-method.

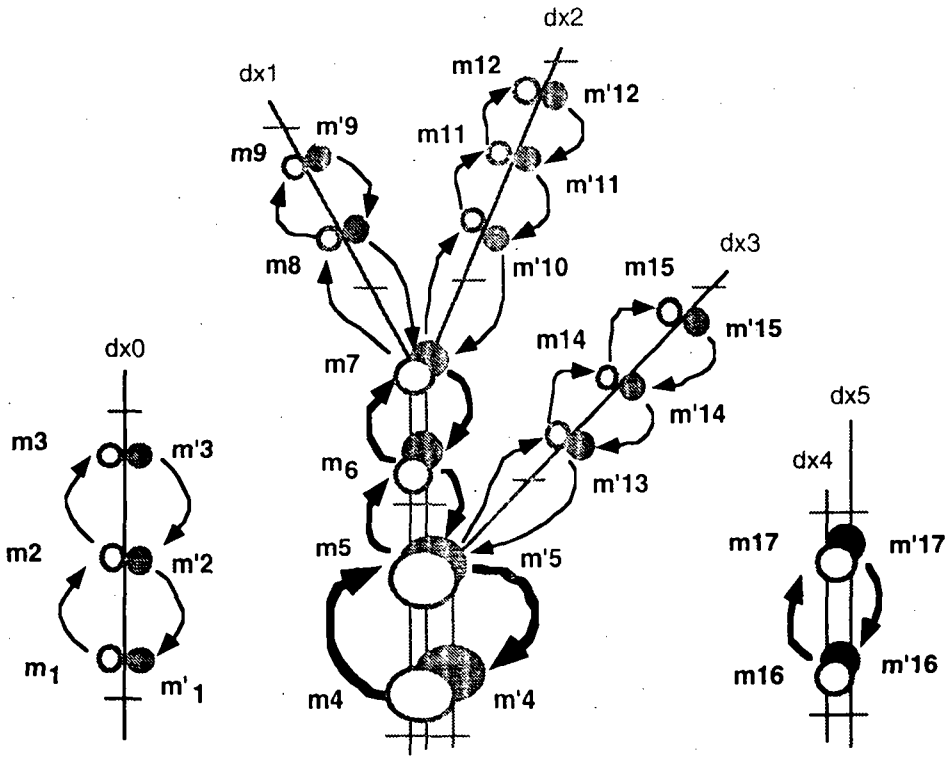


Figure 175.

As explained above, we can either process the whole invocation sequence diagram, or each group or each alley in turn, first for the pre-methods andthen for the post-methods. In the first case (default), this consists in running the combined method shown in figure 173, then the combined method shown in figure 174 : *[[m1 m2 m3] ... [m16 m17] [m'3 m'2 m'1] ... [m'17 m'16]]*. In the second case, the three [combined] methods inside these two global combined methods are interleaved : *[[m1 m2 m3] [m'3 m'2 m'1] ... [m16 m17] [m'17 m'16]]*. In the third case, each list between brackets for pre-methods is completed with corresponding post-items. In any case, the result is the same (if the computation is not interrupted). Next three figures express these three possibilities. (Note that the simplifications in notation implicitely introduced in the formulas, for example [a] [b] => [a b], impose some requirements on the [combination] or the {combination}.)

```
combined method 1 => {   [ m1 m2 m3   ]
                         [ m4 m5  { [ m6 m7  { [ m8 m9 ] [ m10 m11 m12   ] } ]
                                    [ m13 m14 m15    ] } ]
                         [ m16 m17  ]
                    [      m'3 m'2 m'1   ]
                         [ { [ { [ { [ m'9 m'8  ] [ m'12 m'11 m'10    ] } m'7 m'6  ]
                               [ m'15 m'14 m'13    ] } m'5 m'4    ]
                         [ m'17 m'16   ] } .
```

Figure 176.

```
combined method 2 => {   [ m1 m2 m3   m'3 m'2 m'1   ]
                         [ m4 m5  { [ m6 m7  { [ m8 m9 ] [ m10 m11 m12   ] } ]
                                    [ m13 m14 m15    ] }
                           { [ { [ { [ m'9 m'8  ] [ m'12 m'11 m'10    ] } m'7 m'6  ]
                               [ m'15 m'14 m'13    ] } m'5 m'4    ]
                         [ m16 m17   m'17 m'16   ] }
```

Figure 177.

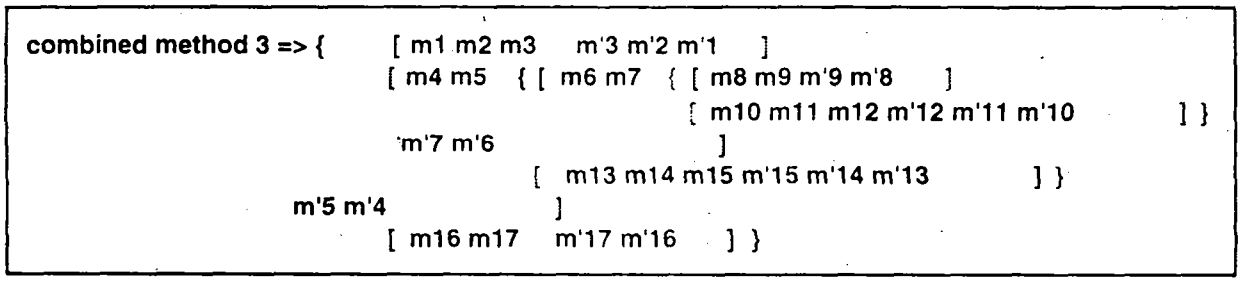


Figure 178

The case just studied corresponds, notably, to the use of primary methods. If we now consider combination methods using qualifiers, once again, due to the independence of each dimension, a number of equivalent processings are possible. We can either process the whole invocation sequence diagram, or each group or each alley in turn, first for the pre-methods (in order : around, before, primary methods) and then for the post-methods (in order : primary, after, around methods). Mixed solutions are also possible. Using the same notation as above, the default solution is : *around-p-combined, before-combined, primary-p-combined, primary-P-combined, after-combined, around-P-combined.* This corresponds to what is done in CLOS (see figure 172) with the methods numbered 1, 2,... to 11, 12... Were the invocation sequence diagram reduced to a single degenerated tree (one or several parallel dimensions), the result would be identical to CLOS.

b) Multiple dispatch

This case derives directly from the above study about single dispatch (§a) since the algorithm for producing the combined method in case of multiple dispatch simply consists in reducing the problem to the case of single dispatch. As explained in subsection 9.2.4 :

(1) the invocation sequence diagram for the *CX0* hierarchy (the hierarchy associated to the first argument) is first ordered as a list of trees : the same thing was done in case of single dispatch ;

(2) the algorithm which is used for the combination of the selected *m* methods —which are multiple-dispatched— is the same than for single-dispatched ones ; and, in both cases, the ordering which is taken into account is precisely the one in the *CX0* invocation sequence diagram.

The refinement for ordering several methods attached to a same ancestor of *CX0* in case of multiple dispatch has no effect on this demonstration ; but it ensures that the result would be identical to the CLOS one in case each *CX0, CY0,...* hierarchy is made of one dimension only.

Hence, sophisticated combinations methods (using qualifiers) may be set up, mimicking and generalizing the CLOS ones, even in case of multiple dispatch.

CONCLUDING PART

11. SUMMARY

COP, an offspring of traditional OOP, is based on the intuition that the basic idea of OOP (*"let's give responsability to data structures"*) is an excellent idea which has not been pushed to its ultimate. COP considers that objects have states and it makes their behaviour depends on these sates (in addition to their classes).

Companion paper n°1 describes the formalism we use for this, i.e. color graphs describing the whole behaviour of a class of objects using states and transitions between these states. A color graph is an **abstract** formalism capturing the states of a class of objects in a N-dimension space, the possible evolution of object states in this space being specified by "regular transitions" triggerred by external events —messages (single dispatch) or generic function calls (multiple dispatch)— and "reflex transitions", i.e. transitions that fire on internal conditions. In addition to being abstract, a color graph describes the **whole** behaviour of a class of objects, i.e. the behaviour implemented by methods and memory representations in this class and its ancestors.

Thus several problems are posed : (inheritance) relationship between a class and its ancestors in terms of color graphs, mapping of a color graph to an implementation (methods and memory representations) ; (inheritance) relationship between an implementation of a color graph and the implementations of its ancestors.

In this paper, inheritance is progressively and fully analysed. Two levels are distinguished :
— the local level (a single color graph is considered) ;
— and the class level (a hierarchy of classes is considered, each one being described by a color graph).
In addition, inheritance is analysed first for transitions (at both levels), then for implementations (at both levels, too).

The basis for inheritance is our local rule for inheritance of transitions : in one given color graph, "regular transitions" are inherited along "reflex transitions". This was shown first in a c-graph (one dimension) —the simplest form of color graph, then in a p-graph (several dimensions). Local inheritance was then extended to class inheritance
Concerning the implementation aspects, a few decisions were taken : a node in a color graph may be attached a memory representation ; a transition, a pre- and a post-method. On that basis, local inheritance of methods and memory representations also along reflex transitions was formulated. This being handled, its extension to class inheritance (in a hierarchy of color graphs) was examined.

The key point in our proposal is to consider objects as being **multidimensional**. Inheritance is viewed in a space of N-dimensions. A critical pass consists in linearizing the classes found along each dimension when considering what an object of a certain class and in a certain states inherits from its class hierarchy. This linearization along several dimensions leads to the definition of **congruency**, a desired property of linearization algorithm : if an algorithm is congruent, then the linearization along one dimension can be obtained —in the correct order— from the linearization of the whole hierarchy by restricting it to classes that impact the dimension in question. The LOOPS linearization algorithm is shown to be congruent. However, it is also shown that congruency and **monotonicity** are antagonistic properties (at least if progression along a path of classes is done blindly). The solution we choose consists in using a monotonic algorithm (we propose a new one) and solving a possible conflict by privileging the leftmost dimension. On the way, we propose a new monotonic linearization algorithm.

Another important property is introduced : **regularity**. Regularity is obtained for memory representations in normal conditions. Concerning methods, a hierarchy is regular vs. its methods named m, if all m methods of degree$\geq K$ that exist along a same dimension do satisfy the same K dimensions. In case of multiple dispatch, the dimensions of all arguments are to take into account. Regularity vs. methods, although not automatic, is obtained in practice. This makes the "invocation sequence diagram" (an abstract of the methods found along the dimensions of a hierarchy) a list of trees (provided that the "prevalence of combined items", a natural rule, is applied). This enables in turn a systematic combination style to be set up, possibly with qualifiers, possibly with multiple dispatch, thus generalizing in all points the CLOS algorithm.

The paper also shows that these very sophisticated method combinations may be obtained without the need of the anti-modular OO construct **send-super** (**super** in Smalltalk, **call-next-method** in CLOS,...), hence a purely declarative mechanism. Concept simplicity is increased in other ways too : individual methods can be ridden of ; the execution order of **:before**, **:after**, ... methods, being based on a more fundamental order, the one of pre- and post-methods, appears more "natural" than otherwise (in CLOS, for example).

12. RELATED WORK

As a description formalism, color graphs are cousins of higraphs [Harel, 1987] [Harel, 1988]. These are used for describing reactive systems. Color graphs take advantage of connectedness whereas higraphs use insideness. The power of expression for non reactive systems is the same. Color graphs can be judged as potentially dangerous since they enable the user to specify blends ; however, this feature is only an alternative enabling simpler specifications in usual cases (it avoids clauses), a mere possibility that is supposed to be used cleverly (when leading to too many blends, users are naturally supposed to refrain from their use).

Predicate classes [Chambers, 1993] also feature states, but the basic mechanism is subclassing : if the state of an instance of class *C* is such that it (dynamically) verifies the predicate of a subclass of *C*, then it inherits automatically from that subclass (cells and methods). Color graphs are more powerful because the next state after a message is basically unknown in predicate classes whereas color graphs specify the destination of each transition.

As mentionned in companion paper n°2, [McGregor-Dyer, 1993] is a work close to us in its basic ideas, yet largely not so fully worked out in its development (even for mixins). It is quite striking to note that the authors basically have the same understanding as ours : they propose two *"techniques"* : *"First, a state from a base class could be decomposed into two or more substates in the derived class. The second technique was to add a new set of states that are in parallel to those that existed in the base classes"* (p. 68). This is akin to the idea of derivation and decomposition. Although with a different formalism, our work is more precise. This was shown about the derivation and mixin constructs, the less easy parts from our point of view (handling orthogonal composition is much easier). Note our work respect the so-called "strict inheritance model" since all the constraints mentioned in the cited work are fulfilled in oour approach. Namely (pp. 64-65) : (1) *"A child class can not delete a state of any of its parent classes"* ; (2) *"Any new state introduced in a child class is wholly contained in a existing state of one of the parent classes"* ; (3) *"A child class may not delete a transition from the state machine of one of its parent classes"* This work does not extract mixins as such : it does not attempt to express an algorithm for combining items found in a whole hierarchy.

A more recent paper is [Sane-Campbell, 1995]. It seems to be much less developed than our work and very much implementation concerned (C++ oriented). As ours, it does not consider concurrency. Mixins are apparently not considered, nor multiple dispatch.

The work which inspired us the most is certainly CLOS [Bobrow et alii, 1988]. The idea of linearizing according to each dimension is directly inspired from CLOS, which does not consider the existence of dimensions, but which linearizes a class hierarchy. The concepts of qualifiers for building sophisticated combinations and MOPs for enabling some changes vs. the proposed defaults are those of CLOS. Our work extends the CLOS proposal by taking into account states. The proposal we make enables the elimination of the **call-next-method**, an anti-modular construct the use of which is discouraged in CLOS : COP thus features a purely declarative style as recommended by Sonya Keene [Keene, 1989]. Note also the disparition of the individual methods, i.e. specialized by an (**eql** form).

As far as the linearization is concerned, we deeply appreciate the work done in [Ducournau et alii, 1992], [Ducournau et alii, 1994] and [Ducournau et alii, 1995] as well as other papers from the same authors. The rigor they introduce in this domain is welcome. We modestly attempt to define congruency in the same way. A number of questions remain open : is the LOOPS linearization algorithm the only one that is blind and congruent ? Is it possible or impossible to find a non blind algorithm that is both congruent and monotonic ? Would it be useful ? (i.e. is a better solution than the one presented here possible ?)

Concerning inheritance, it seems -to the best of our knowledge- that our approach "by dimensions" is completely new. One can interpret our work as somehow providing automatically a correct "point of view" [Carré-Geib, 1990] (here, a dimension) using the color graph description (a number of naming problems are avoided due to narrower contexts ; this is not to say that all naming problems disappear : the solution proposed by the authors is interesting in this respect).

We are currently browsing the literature to test COP against what may provoke difficulties in OOP. We found a solution to the so-called "state partitioning anomaly" as defined in [Aksit-Bergmans, 1992]. A separate paper will describe it in details. It also appears that the proposal made by [Ossher-Harrison, 1992] for combining inheritance hierarchies meshes well with our approach by dimensions with the decomposition and derivation constructs.

13. CONCLUSION

In this paper, inheritance is progressively developed from simple and intuitive local inheritance rules in a single color graph for transitions up to a highly sophisticated combination algorithm for multiply-dispatched methods (in fact, the most sophisticated we know) that fully generalizes the CLOS approach. Both a systematic masking style ("Smalltalk-style") and a systematic combination style ("CLOS-style") are addressed. Are also addressed, orthogonally, both a message-style (single dispatch) and a generic-function-style (multiple dispatch). Contrasting with the imposition of magic rules extracted out of a hat like a bunny rabbit, this development fully explains all inheritance rules.

Given that the color graph formalism we use is "connectedness-oriented" whereas the higraph formalism is "insideness-oriented", our results may be incorporated in the statechart formalisms and derived ones.

From a conceptual point of view, the interpretation of inheritance in terms of dimensions was found quite illuminating. It may well bring new and important results due to a new mathematical point of view on inheritance. Intuitively, it appears that merging all the points (coordinates) that exist on one dimension, as it is implicitely done in traditional OOP, certainly makes inheritance difficult to deal with.

BIBLIOGRAPHY

[Aksit-Bergmans, 1992] Mehmet Aksit & Lodewijk Bergmans. *"Obstacles in Object-Oriented Development"*. In OOPSLA'92 Proceedings. October 18-22, 1992.

[America, 1990] Pierre America. *"Designing an Object-Oriented Programming Language with behavioural subtyping"*. Lecture Notes in Computer Science no 489. pp.60-90. May-June 1990.

[Bobrow et alii, 1988] Daniel G. Bobrow , Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales & David A. Moon. *"CLOS"*. X3J13 Document 88-002R. June 1988.

[Borron, 1995a] Henry J. Borron. *"Colored-Object Programming : (1) Describing Interfaces"*. In GL'95 Proceedings. (Huitièmes Journées Internationales sur le Génie Logiciel et ses Applications.) November 15-17, 1995.

[Borron, 1995b] Henry J. Borron. *"Colored-Object Programming : (2) Concrete and Abstract Implementations"*. In GL'95 Proceedings. November 15-17, 1995.

[Borron, 1996a] Henry J. Borron. *"Colored-Object Programming : Goals and Metaphors"*. January 1996.

[Borron, 1996b] Henry J. Borron. *"Colored-Object Programming : About the visual formalism"*. (First version in January 1996.) Research Report 2879, april 1996.

[Borron, 1996c] Henry J. Borron. *"Colored-Object Programming : About the programming activity"*. (First version in January 1996.) Research Report 2880, april 1996.

[Borron, 1996d] Henry J. Borron. *"Colored-Object Programming : Color graphs, a visual formalism for synthesizing the behaviour of objects"*. (First version in February 1996.) Research Report 2876, april 1996.

[Borron, 1996e] Henry J. Borron. *"Colored-Object Programming : mixin and derivation, two conjoint concepts for a rigorous handling of independent supplementary behaviours"*. (First version in February 1996.) Research Report 2877, april 1996.

[Borron, 1996h] Henry J. Borron. *"Colored-Object Programming : Ergonomic and cognitive issues"*. May 1996. To be published in ERGO-IA'96 Proceedings (october 1996).

[Borron, 1996x] Henry J. Borron. *"A two-pass effcient and monotonic linearization algorithm"*. In preparation. INRIA Sophia-Antipolis. 1996.

[Carré-Geib, 1990] Bernard Carré & Jean-Marie Geib. *"The Point of View notion for Multiple Inheritance"*. In ECOOP/OOPSLA'90 Proceedings. October 21-25, 1990.

[Chambers, 1993] Craig Chambers. *"Predicate classes"*. In ECOOP'93 Proceedings. pp. 268-296. Springer-Verlag. July 1993.

[Ducournau et alii, 1992] R.Ducournau, M. Habib, M. Huchard & M.L.Mugnier. *"Monotonic conflict resolution for inheritance"*. In OOPSLA'92 Proceedings. October 18-22 1992. pp. 16-24.

[Ducournau et alii, 1994] R.Ducournau, M. Habib, M. Huchard & M.L.Mugnier. *"Proposal for a monotonic multiple linearization"*. In OOPSLA'94 Proceedings. October 23-27 1994. pp. 164-175.

[Ducournau et alii, 1995] R.Ducournau, M. Habib, M. Huchard, M.L.Mugnier & A. Napoli. *"Le point sur l'héritage multiple"*. In TSI. Vol. 14, n°3. 1995. pp. 309-345.

[Green, 1982] T.R.G. Green. *"Pictures of programs and other processes, or how to do things with lines"*. Behaviour and Information psychology. Vol. 1, no 1, pp. 3-36. 1982.

[Harel, 1987] David Harel. *"Statecharts : a visual formalism for complex systems"*. Science of Computer Programming. Volume 8, no 3, pp. 231-274. June 1987.

[Harel, 1988] David Harel. *"On visual formalisms"*. Communications of the ACM, Vol. 31, no 5, pp. 514-530. May 1988.

[Harel et alii, 1987] D. Harel, A. Pnueli, J.P. Schmidt, R. Scherman. *"On the formal semantics of statecharts"*. (Extended abstract). In proceedings of the second IEEE symposium on logic in computer science (Ithaca). pp. 54-64. IEEE Press. June 1987.

[Keene, 1989] Sonya E. Keene. *"Object-oriented progamming in Common Lisp. A programmer's guide to CLOS"*. Addison-Wesley. 1989.

[Kiczales et alii, 1992] Gregor Kiczales, Jim des Rivières & Daniel G. Bobrow. *"The Art of the Metaobject Protocol"*. MIT Press, 1992.

[McGregor-Dyer, 1993] John D. McGregor & Douglas M. Dyer. *"A note on inheritance and state machines"*. Software Engineering Notes, Vol 18, no 4. October 1993.

[Ossher-Harrison, 1992] Harold Ossher & William Harrison. *"Combination of Inheritance Hierarchies"*. OOPSLA. 1992.

[Sane-Campbell, 1995] Aamod Sane & Roy Campbell. *"Object-oriented machines: subclassing, composition, delegation and genericity"* . OOPSLA. 1995.

APPENDIX

THE EXPANSION OF STQ

Next figure shows the expansion of the *STQ* p-graph. Instead of *not empty*, we note *q* (resp. *s*) the substate corresponding to "the number of pushed (resp. enqueued) elements is strictly positive". The graph is not symmetrical because of the masking effect of *pop* in *Stack*.

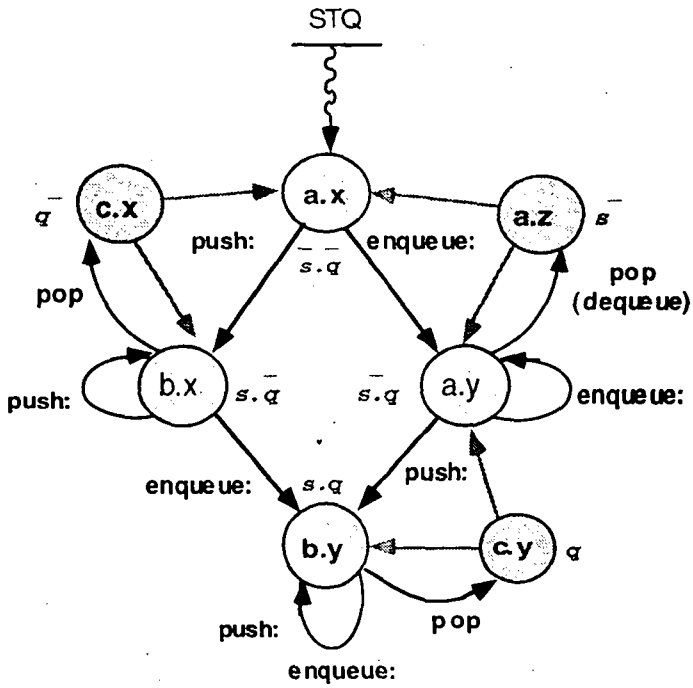


Figure A.1.

(Note : This figure is not meant to be interacted with by the programmer. In fact, our proposal precisely enables the programmer to avoid building it in a brutal way. This figure simply shows that a relatively complex result may be obtained with a minimum of effort.)

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine - Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes - IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes - 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de recherche INRIA Rocquencourt - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 6399